Hyperblock Scheduling for Verified High-Level Synthesis

YANN HERKLOTZ, Imperial College London, United Kingdom
JOHN WICKERSON, Imperial College London, United Kingdom

High-level synthesis (HLS) is the automatic compilation of software programs into custom hardware designs. With programmable hardware devices (such as FPGAs) now widespread, HLS is increasingly relied upon, but existing HLS tools are too unreliable for safety- and security-critical applications. Herklotz et al. partially addressed this concern by building *Vericert*, a prototype HLS tool that is proven correct in Coq (à la CompCert), but it cannot compete performance-wise with unverified tools. This paper reports on our efforts to close this performance gap, thus obtaining the first *practical* verified HLS tool. We achieve this by implementing a flexible operation scheduler based on *hyperblocks* (basic blocks of predicated instructions) that supports *operation chaining* (packing dependent operations into a single clock cycle). Correctness is proven via translation validation: each schedule is checked using a Coq-verified validator that uses a SAT solver to reason about predicates. Avoiding exponential blow-up in this validation process is a key challenge, which we address by using *final-state predicates* and *value summaries*. Experiments on the PolyBench/C suite indicate that scheduling makes Vericert-generated hardware 2.1× faster, thus bringing Vericert into competition with a state-of-the-art open-source HLS tool when a similar set of optimisations is enabled.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Hardware** → **Operations scheduling**; • **Theory of computation** → *Program verification.*

Additional Key Words and Phrases: Coq, CompCert, translation validation, operation chaining, symbolic evaluation

ACM Reference Format:
Yann Herklotz and John Wickerson. 2024. Hyperblock Scheduling for Verified High-Level Synthesis. *Proc. ACM Program. Lang.* 8, PLDI, Article 225 (June 2024), 25 pages. https://doi.org/10.1145/3656455

1 INTRODUCTION

High-level synthesis (HLS) is the automatic compilation of software programs into custom hardware designs. With programmable hardware devices (such as FPGAs) now widespread, HLS tools such as AMD Vitis HLS [4], Intel HLS Compiler [30], LegUp [12], and Bambu [19] are increasingly relied upon. But existing HLS tools are too unreliable for safety- and security-critical applications; indeed, random testing uncovered miscompilations in all four of the above tools [24].

Herklotz et al. [25] have begun to address this shortcoming by building *Vericert*, a prototype HLS tool that extends the CompCert verified C compiler [33]. Vericert guarantees reliability by being proven correct in Coq, but unfortunately, the hardware designs it generated could not compete performance-wise with those generated by unverified HLS tools. Its key weakness was that it serialised every operation, taking no advantage of the parallelism that custom hardware offers.

This paper reports on our efforts to close this performance gap by extending Vericert with a scheduling pass that collects operations into groups that can be executed in parallel.

Authors' addresses: Yann Herklotz, Imperial College London, London, United Kingdom, yann.herklotz15@imperial.ac.uk; John Wickerson, Imperial College London, London, United Kingdom, j.wickerson@imperial.ac.uk.

ACM 2475-1421/2024/6-ART225
https://doi.org/10.1145/3656455

Context. Many approaches to scheduling have been proposed over the years. Some, such as *list scheduling* [5, p. 257] only reorder instructions within a basic block. This means they squander opportunities for performance improvements that could be obtained by reordering instructions across branches. A more powerful alternative is *trace scheduling* [17, 20], which works by creating paths (or 'traces') through the code, across basic block boundaries, and then reorders the instructions within those paths. In its most general form, trace scheduling is considered infeasible on large programs, but two special cases called *superblock scheduling* [28] and *hyperblock scheduling* [34] have been developed, both of which impose restrictions on the form of traces in order to obtain tractable algorithms. Superblocks generalise basic blocks by allowing early exits, while hyperblocks generalise superblocks by additionally allowing each instruction to be *predicated*. CPUs often lack support for predicated execution, so superblock scheduling is the natural choice in that setting. But in custom hardware, predicated execution can be implemented efficiently, making hyperblock scheduling a natural fit for HLS. Indeed, the use of hyperblock scheduling in HLS was first proposed over two decades ago [9, 10], and is nowadays used by popular HLS tools such as AMD Vitis HLS [3], LegUp [11, p. 60], Google XLS [21, line 112], and Bambu [18, line 304]. Hyperblock scheduling is therefore a natural choice for Vericert to close the gap between the verified and unverified HLS tools, and model what existing tools are already doing. Support for the scheduling of predicated instructions is also important for future HLS-specific optimisations, such as loop scheduling, because these often assume that basic blocks can be merged using if-conversion.

Contributions. In this paper, we present:

- **the first verified implementation of hyperblock scheduling**, which is more general than Six et al.'s verified superblock scheduler [40] and more computationally tractable than Tristan and Leroy's verified trace scheduler [43],
- **the first verified implementation of general if-conversion** (a pre-scheduling pass that turns if-statements into hyperblocks), building on CompCert's naïve if-converter that only handles simple cases [1],
- **a novel use of a verified SAT solver during translation validation** in order to reason about predicates, and
- **experiments on the widely used PolyBench/C suite** [36] showing that scheduling makes Vericert generate 2.1× faster hardware, thus making it competitive with Bambu [19], a state-of-the-art open-source HLS tool, when a similar set of optimisations is enabled.

2 OVERVIEW

Our starting point is the first version of Vericert [25], which generates strictly sequential hardware designs: each clock cycle has just a single C operation mapped to it.

Making those hardware designs parallel is actually not the challenging part of our work – Verilog synthesis tools do this implicitly. Indeed, it is easy to write Verilog so that an arbitrary sequence of C operations is mapped to a single piece of combinational logic that executes in just one clock cycle. But the problem with such unfettered parallelism is that these large pieces of combinational logic may have long critical paths, and thus lead to hardware that can only run at a low clock frequency. The actual challenge, then, is producing the *right amount* of parallelism.

This is a scheduling problem. Vericert must decide how to schedule each block of instructions across one or more clock cycles so that when the downstream Verilog synthesis tool performs parallelisation, the resulting combinational paths will be short enough to meet the target frequency.

To clarify: the downstream synthesis tool is responsible for the parallelisation itself – Vericert does not perform it, nor does it guarantee its soundness. Rather, it predicts the parallelisation that the synthesis tool will perform, and schedules so that if parallelisation is performed as predicted, the

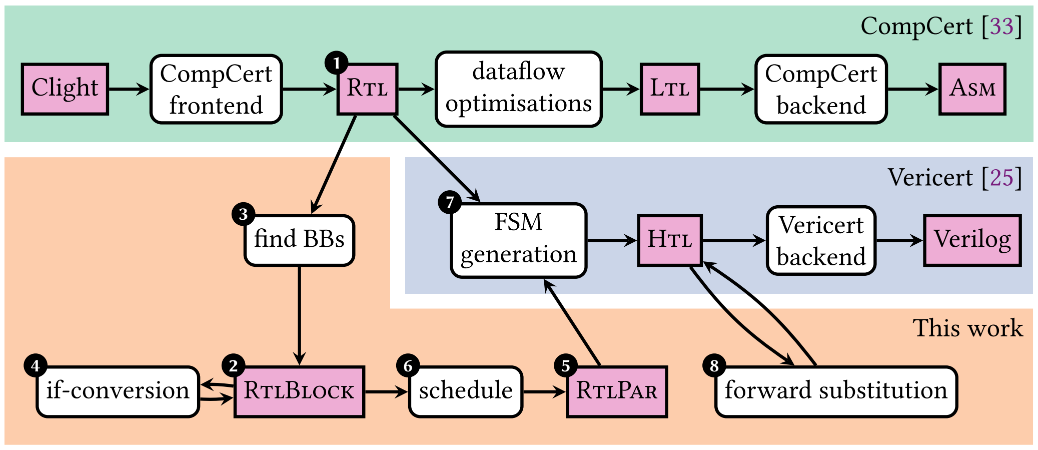


Fig. 1. New passes and intermediate languages introduced in this work.

target frequency should be met. What Vericert *does* guarantee is that if it reorders any instructions as part of the scheduling process, these reorderings preserve the program's (sequential) behaviour.

An overview of our solution is given in fig. 1, which shows the main components of our hyperblock scheduler and how they fit into the wider Vericert and CompCert projects.

❶ We intercept the CompCert flow at the Register Transfer Language (RTL),[1] where each function in the program is represented as a CFG whose nodes are individual three-address instructions. Were we to intercept any earlier in the flow, we would have to construct this CFG ourselves; any later, and register allocation would have been performed. Register allocation is unhelpful for us because we are not targeting an architecture with a fixed set of registers.

❷ We need to identify the basic blocks so they can be combined into hyperblocks for scheduling. (CompCert does already have basic block creation at the Location Transfer Language, LTL, but as mentioned above, Vericert's flow has already branched off at RTL.) Hence, we define a new intermediate language called RTLBLOCK, which is a CFG whose nodes are basic blocks.

❸ We compile from RTL to RTLBLOCK using a translation-validation approach. That is, an unverified algorithm gathers instructions into basic blocks, then a verified validator confirms the equivalence of the RTL and RTLBLOCK representations. This is a simpler version of CompCert's basic block creator for LTL, because no variables are moved to the stack.

❹ We then perform *if-conversion* [2]. This is a transformation that merges basic blocks from two sides of a fork in the CFG into a single, larger basic block (now a hyperblock) that uses predication to control which instructions are executed. If-conversion is helpful because larger blocks can give the scheduler more opportunities to find parallelism. (CompCert does already have an if-conversion pass [1], but it can only handle simple cases such as replacing `if(c){x=a;} else {x=b;}` with `x=c?a:b`, whereas our implementation can handle arbitrary forks in the CFG.) If-conversion can be applied selectively, using heuristics to judge which basic blocks should be combined, and once if-conversion has been proved correct wherever it is applied, these heuristics can be adjusted with no impact on the correctness proof.

❺ We then perform scheduling on each hyperblock. The scheduler takes the list of predicated instructions and produces a list of groups of instructions, such that all the instructions in the same group can be scheduled for the same clock cycle without violating timing requirements.

[1]Note that this is not the RTL that stands for Register-Transfer Level in hardware design.

Actually, our scheduler produces a list of groups of *lists* of instructions. The idea behind this three-level representation, which we call RtlPar, is that each inner list is a sequence of instructions that can be chained together, then each group contains instruction chains that we expect the downstream synthesis tool to place in parallel. This way, we support *operation chaining*, a long-established optimisation in hardware design [35, p. 1101].

❻ The scheduler itself is written in unverified OCaml and works similarly to those in existing HLS tools [12]: it takes a set of scheduling constraints that capture the target clock frequency, available hardware resources, and dependencies between operations, encodes them all as a system of difference constraints (SDC) [15], and hands them off to a linear program solver.

❼ The CFG is then encoded as a finite state machine (FSM) in a Verilog-like language called Hardware Transfer Language (Htl). This process is largely inherited from Vericert, but where Vericert produces FSMs that perform just one assignment per state, we produce FSM states with a sequence of assignments.

❽ We have a final pass that performs *forward substitution* [27, p. 109]. This turns each sequence of Verilog blocking assignments into an equivalent sequence of nonblocking assignments, which makes the downstream logic synthesis tool more likely to perform the parallelisation that our scheduler predicted. For example:

```
r = r1 * r2;
r = r + r3;
r5 = r + r4;
```

$\xrightarrow[\text{substitution}]{\text{forward}}$

```
r <= r1 * r2;
r <= (r1 * r2) + r3;
r5 <= ((r1 * r2) + r3) + r4;
```

The two versions are semantically equivalent, but we find that the second, in which both right-hand sides must be evaluated before either assignment is performed, makes the downstream logic synthesis tools more likely to produce the hardware we intend (which, in this particular example, involves exploiting a fused multiply–accumulator unit if available).[2]

What remains is to ensure the correctness of each schedule produced in step ❻. Following previous work on verified scheduling by Tristan and Leroy [43] and Six et al. [40], we use translation validation, but dealing with hyperblocks brings additional complexity, as explained below.

Tristan and Leroy implement trace scheduling in its full generality. They use a tree to represent all the possible control-flow paths through a block. These trees can be "exponentially larger than the original code" [43, p. 25], which makes it prohibitively expensive to construct and compare the trees before and after scheduling, and thus undermines the usefulness of their scheduler.

Six et al. restrict their scheduler to superblocks. Since a superblock has only a single control-flow path (with early exits), the need for trees is avoided. This allows their validator to be "efficient even for large superblocks" [40, p. 53]. However, superblocks are less general than hyperblocks, and there are code patterns where superblock scheduling can lead to considerable code duplication that hyperblocks would avoid. Moreover, superblock scheduling is reliant on profiling and branch-prediction heuristics to pick a hot path through the program – should such a hot path even exist.

Hyperblocks can branch and merge control-flow using predicated execution, and hence a single hyperblock can capture many control-flow paths without the exponential blow-up that Tristan and Leroy encountered. In particular, hyperblocks can handle well the case where two branches of a conditional statement are executed equally often, unlike the superblocks that Six et al. use. However, our task of validating the equivalence of two hyperblocks is complicated by having to reason about predicates. Where the prior works only needed to check that the scheduled block contains a dependency-respecting permutation of the original block's instructions, we must account for the

[2]As the example shows, forward substitution does not remove duplicate writes to a variable because Verilog semantics explicitly state that the order of nonblocking assignments to the same variable will be preserved [29, p. 254].

$$
\begin{array}{rrcll}
\text{registers:} & r, \texttt{r1}, \texttt{r2}, \ldots \in \mathbb{r} & & & \\
\text{predicates:} & p, \texttt{p1}, \texttt{p2}, \ldots \in \mathbb{p} & & & \\
\text{CFG node labels:} & L \in \mathbb{L} & ::= & \mathbb{N} & \\
\text{guard expressions:} & G \in \mathbb{G} & ::= & \mathbb{p} \mid \neg\mathbb{p} \mid \mathit{true} \mid \mathit{false} \mid \mathbb{G} \land \mathbb{G} \mid \mathbb{G} \lor \mathbb{G} & \\
\text{arithmetic ops:} & op_{\mathrm{a}} \in \mathbb{a} & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \ldots & \\
\text{conditional ops:} & op_{\mathrm{c}} \in \mathbb{c} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \ldots & \\
\text{addressing modes:} & d \in \mathbb{d} & ::= & \texttt{Stack} \mid \texttt{Global} \mid \ldots & \\
\text{instructions:} & I \in \mathbb{I} & ::= & \texttt{skip} & \text{(no-op)} \\
 & & \mid & \mathbb{G}\texttt{ => }\mathbb{r}\texttt{ := }\mathbb{r}\ \mathbb{a}\ \mathbb{r} & \text{(arith/logical op)} \\
 & & \mid & \mathbb{G}\texttt{ => }\mathbb{r}\texttt{ := }\mathbb{d}\texttt{[}\mathbb{r}\texttt{]} & \text{(memory load)} \\
 & & \mid & \mathbb{G}\texttt{ => }\mathbb{d}\texttt{[}\mathbb{r}\texttt{] := }\mathbb{r} & \text{(memory store)} \\
 & & \mid & \mathbb{G}\texttt{ => }\mathbb{p}\texttt{ := }\mathbb{r}\ \mathbb{c}\ \mathbb{r} & \text{(assign predicate)} \\
 & & \mid & \mathbb{G}\texttt{ => E(}\mathbb{I}_{\mathrm{cf}}\texttt{)} & \text{(block exit)} \\
\text{control-flow instructions:} & I_{\mathrm{cf}} \in \mathbb{I}_{\mathrm{cf}} & ::= & \texttt{if (}\mathbb{r}\ \mathbb{c}\ \mathbb{r}\texttt{) }\mathbb{L}\ \mathbb{L} & \text{(conditional)} \\
 & & \mid & \texttt{goto }\mathbb{L} & \text{(goto node)} \\
 & & \mid & \ldots & \\
 & H \in \textsc{RtlBlock} & ::= & I\ \texttt{list} & \\
 & H_{\mathrm{par}} \in \textsc{RtlPar} & ::= & I\ \texttt{list list list} &
\end{array}
$$

Fig. 2. Syntax of RtlBlock and RtlPar, with our hyperblock additions highlighted.

fact that predicates may be modified during scheduling. For instance, the sequence `p => i; !p => i`, which executes `i` if `p` holds and then executes `i` if `p` does *not* hold, may be optimised to `i`.

The approach we take is to translate both the RtlBlock hyperblock and the RtlPar hyperblock (i.e., before and after scheduling) into their strongest postconditions, starting from the same symbolic initial state, and then comparing these postconditions for equivalence with the help of a SAT solver that we have programmed and verified in Coq. By being able to solve queries like $(\texttt{p} \land \texttt{!p}) \leftrightarrow \mathit{false}$, the SAT solver enables reasoning about reordering of instructions in a predicate-aware fashion.

3 NEW INTERMEDIATE LANGUAGES

Our work introduces two new intermediate languages: RtlBlock and RtlPar, which implement the sequential semantics of hyperblocks. They are based on CompCert's Rtl, but instead of mapping from states to instructions, RtlBlock maps from states to hyperblocks, and RtlPar maps from states to hyperblocks of instructions grouped into specific cycles.

Hyperblocks are made up of instructions as defined in fig. 2, where $\vec{\cdot}$ denotes a list and $\cdot^{?}$ denotes an optional parameter. Most instructions are similar to their Rtl counterparts, except each instruction is now guarded by an optional predicate. One additional instruction is for setting a predicate (p) equal to an evaluated condition ($r_1\ op_{\mathrm{c}}\ r_2$). The other new instruction is E, which takes a control-flow instruction (I_{cf}) and allows for early exit from the hyperblock.

These instructions are used in both RtlBlock and RtlPar. The main difference between these two languages is how these instructions are arranged within the hyperblock and the execution semantics of the hyperblock. An RtlBlock hyperblock is a list of instructions, with a straightforward sequential semantics. An RtlPar hyperblock is a list of lists of lists of instructions, with nested blocks corresponding to where instructions should be placed in hardware. Each innermost list contains a chain of instructions that can be executed sequentially within a single clock cycle; each middle list contains a group of chains that can be executed in parallel; and the outermost list contains groups to be executed sequentially (in consecutive clock cycles).

$$\textsc{ExecInstr}\ \frac{\Gamma_{\mathrm{P}} \vdash G \downarrow \mathit{true} \qquad \Gamma \vdash \texttt{r1}\ op_{\mathrm{a}}\ \texttt{r2} \downarrow v}{\Gamma \vdash (G \texttt{ => } \texttt{rd} \texttt{ := } \texttt{r1}\ op_{\mathrm{a}}\ \texttt{r2}) \Downarrow_{\mathbb{I}} ((\Gamma_{\textsc{Env}}, \Gamma_{\mathrm{R}}[\texttt{rd} \mapsto v], \Gamma_{\mathrm{P}}, \Gamma_{\mathrm{M}}), \mathsf{None})} \qquad \textsc{ExecInstrFalse}\ \frac{\Gamma_{\mathrm{P}} \vdash G \downarrow \mathit{false}}{\Gamma \vdash (G \texttt{ => } \_) \Downarrow_{\mathbb{I}} (\Gamma, \mathsf{None})}$$

$$\textsc{ExecExit}\ \frac{\Gamma_{\mathrm{P}} \vdash G \downarrow \mathit{true}}{\Gamma \vdash (G \texttt{ => } \texttt{E}(I_{\mathrm{cf}})) \Downarrow_{\mathbb{I}} (\Gamma, \mathsf{Some}(I_{\mathrm{cf}}))} \qquad \textsc{BlockContinue}\ \frac{\Gamma \vdash I \Downarrow_{\mathbb{I}} (\Gamma', \mathsf{None}) \qquad \Gamma' \vdash l \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma'', I_{\mathrm{cf}})}{\Gamma \vdash I :: l \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma'', I_{\mathrm{cf}})}$$

$$\textsc{BlockExit}\ \frac{\Gamma \vdash I \Downarrow_{\mathbb{I}} (\Gamma', \mathsf{Some}(I_{\mathrm{cf}}))}{\Gamma \vdash I :: l \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma', I_{\mathrm{cf}})} \qquad \textsc{ExecRtlBlock}\ \frac{\Gamma \vdash H \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma', I_{\mathrm{cf}})}{\Gamma \vdash H \Downarrow_{\textsc{RtlBlock}} (\Gamma', I_{\mathrm{cf}})} \qquad \textsc{ExecRtlPar}\ \frac{\Gamma \vdash \mathsf{concat}(\mathsf{concat}\ H_{\mathrm{par}}) \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma', I_{\mathrm{cf}})}{\Gamma \vdash H_{\mathrm{par}} \Downarrow_{\textsc{RtlPar}} (\Gamma', I_{\mathrm{cf}})}$$

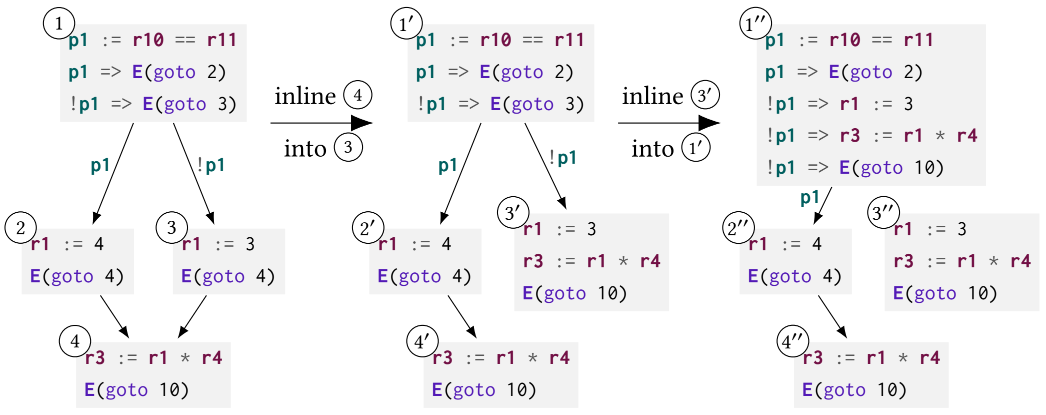
Fig. 3. Semantics of RtlBlock and RtlPar hyperblocks, where Γ is a 4-tuple $(\Gamma_{\textsc{Env}}, \Gamma_{\mathrm{R}}, \Gamma_{\mathrm{P}}, \Gamma_{\mathrm{M}})$.

The existing CompCert semantics for Rtl is a small-step operational semantics defined on a CFG. At each step, the instruction in the CFG at the current program counter is evaluated in a context Γ. This is a 3-tuple comprising an environment $\Gamma_{\textsc{Env}}$ that has global information about the program, a mapping Γ_{R} from registers to values, and a mapping Γ_{M} from memory addresses to values.

In order to give a semantics for RtlBlock and RtlPar, we need to handle predicates, so we turn Γ into a 4-tuple $(\Gamma_{\textsc{Env}}, \Gamma_{\mathrm{R}}, \Gamma_{\mathrm{P}}, \Gamma_{\mathrm{M}})$, where the additional component Γ_{P} maps predicates to Booleans. Moreover, we need to deal with CFGs where each node is not just a single instruction, but a hyperblock. The semantics, which we provide in fig. 3, is denoted by $\Gamma \vdash a \Downarrow_A b$ and states that under the context Γ and using the definition A, a will be evaluated to b. It is a big-step semantics in the sense that it executes an entire hyperblock in a single step. The ExecInstr rule executes an arithmetic instruction if its guard evaluates to true (and there are similar rules for the other guarded instructions). Here we write $\Gamma \vdash a \downarrow b$ for an evaluation function for operations: given Γ and a it computes b. The ExecInstrFalse rule handles the case where the guard does not hold. ExecExit handles the execution of an exit instruction by recording the control-flow instruction I_{cf} for leaving the block. The next two rules are for executing an instruction list, with BlockContinue handling the case where the head instruction does not exit the block, and BlockExit handling the case where it does. Finally, ExecRtlBlock and ExecRtlPar provide the semantics of RtlBlock and RtlPar blocks respectively. Both languages use the list execution semantics defined by the above rules, but RtlPar blocks are first flattened into a single list (via concat).

Note that these rules define a sequential semantics for RtlPar. That is, although RtlPar blocks contain lists of instruction chains that have been identified by the scheduler as being suitable for parallel execution, our semantics nonetheless executes them sequentially. This is because although the scheduler identifies where parallelism can be profitably extracted, it does not perform any parallelisation itself – this is left for the logic synthesis tool. (A parallel RtlPar semantics may allow more optimisations to be validated, but we save that for future work.)

The overall behaviour $\mathcal{B}$ of an RtlBlock or RtlPar program is the same as that of existing CompCert intermediate languages. Either the program terminates with a return value and a finite trace of externally visible events, like external I/O events which can be produced by system calls (part of the I_{cf} instructions), or the program diverges with a possibly infinite trace of external events. Note that the top-level correctness theorem of Vericert does not include a trace of externally visible events, because the only external behaviour of the hardware is the final return value; nevertheless, until the hardware is generated, instructions emitting external behaviours are kept and reasoned about.



```
p1 := r10 == r11
p1 => E(goto 2)
!p1 => E(goto 3)

r1 := 4
E(goto 4)

r1 := 3
E(goto 4)

r3 := r1 * r4
E(goto 10)
```

```
p1 := r10 == r11
p1 => E(goto 2)
!p1 => E(goto 3)

r1 := 4
E(goto 4)

r1 := 3
r3 := r1 * r4
E(goto 10)

r3 := r1 * r4
E(goto 10)
```

```
p1 := r10 == r11
p1 => E(goto 2)
!p1 => r1 := 3
!p1 => r3 := r1 * r4
!p1 => E(goto 10)

r1 := 4
E(goto 4)

r1 := 3
r3 := r1 * r4
E(goto 10)

r3 := r1 * r4
E(goto 10)
```

Fig. 4. An example showing two iterations of the block inlining pass.

4 VERIFIED IF-CONVERSION

If-conversion introduces the predicated instructions that form hyperblocks. As mentioned in section 2, CompCert does have an if-converter already, but it applies only in a few special cases. We need a more general algorithm that can handle arbitrary branching code. We therefore implement the first formally verified implementation of a general if-conversion algorithm, with support for heuristics for branch prediction. To simplify both the implementation of the if-converter and its correctness proof, it is split up into three distinct transformations:

(1) **Condition Elimination** First, every conditional instruction in the block is replaced by two predicated goto instructions. For example:

```
E(if c n1 n2);
```

$$\xrightarrow[\text{elimination}]{\text{condition}}$$

```
p := c;
p => E(goto n1);
!p => E(goto n2);
```

(2) **Block Inlining** This is performed by replacing a predicated goto instruction by the list of instructions in the block that it points to, adding the predicate to each of those instructions. This transformation only converts one level of blocks at a time, but it can be repeatedly invoked to create larger blocks, as shown in fig. 4. The pointed-to block is left unchanged in case it is still pointed to by other blocks; as such, this transformation performs *tail duplication* [13].

(3) **Dead Block Elimination** Finally, any blocks that are now unreachable from the function's entry-point (such as 3″ in fig. 4) are removed, to reduce code size.

The decision about which goto instructions should be inlined is offloaded to an external procedure. This separation of concerns means that the correctness of the transformation can be proven once-and-for-all for a single, general, if-conversion algorithm, which can then be extended with various heuristics to change the performance of the generated code. In our implementation, we use simple static heuristics to pick these paths, following Ball and Larus [6], such as avoiding inlining loop back-edges, or blocks with an instruction count that exceeds a threshold (currently 50).

To prove the top-level end-to-end correctness theorem of Vericert, forward simulations proven for each transformation are composed together. The following theorem is a forward simulation and states the correctness of if-conversion.

Theorem 4.1 (Forward simulation of if-conversion). *If program S is* `safe` *(free from undefined behaviour) and has behaviour $\mathcal{B}$, then* `ifconvert`(S) *should have the same behaviour. That is:* $\texttt{safe}(S) \wedge S \Downarrow \mathcal{B} \implies \texttt{ifconvert}(S) \Downarrow \mathcal{B}$.

Proof sketch. Each of the three transformations is verified using the simulation-diagram approach [33, p. 379]. These simulations are then composed into an overall simulation for if-conversion. Condition elimination is straightforward because it is a purely local replacement. Dead block elimination is also straightforward, being similar to a CFG-pruning transformation from CompCertSSA [7]. The block inlining pass is a bit more involved. To see why, consider how to prove a forward simulation for the first transformation in fig. 4. The edges $\textcircled{1} \to \textcircled{1'}$, $\textcircled{2} \to \textcircled{2'}$, and $\textcircled{4} \to \textcircled{4'}$, are straightforward, but $\textcircled{3}$ is tricky, because $\textcircled{3'}$ does not straightforwardly simulate $\textcircled{3}$ (there is no edge from $\textcircled{3'}$ that can mimic the edge from $\textcircled{3}$ to $\textcircled{4}$). To resolve this, we make our simulation relation a little more fine-grained, so that $\textcircled{3}$ can be mapped to the first 'part' of $\textcircled{3'}$ and $\textcircled{4}$ can be mapped to the second. □

5 IMPLEMENTING HYPERBLOCK SCHEDULING

This section discusses our implementation of hyperblock scheduling in Vericert. The scheduler takes each hyperblock of an if-converted RtlBlock program in turn, and schedules it to form an RtlPar hyperblock. The scheduler is unverified, but it uses a verified translation validation algorithm to prove each output correct, which we will describe in section 6.

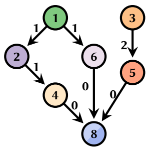
Our scheduler is written in OCaml, and follows the SDC scheduling approach [15]. SDC scheduling is widely used in HLS tools, and it has also been extended to support modulo scheduling of loops [47], which we hope to incorporate into our future work. The SDC scheduler generates a function that should be minimised plus a set of constraints that must be respected while doing so. In our case, the function we minimise is the overall latency of the block (i.e. the end time of the last operation). The constraints come from three sources. First, the cumulative latency of all the operations in each chain must not exceed a predefined limit; this ensures that operation chaining does not reduce the maximum clock frequency of the resultant hardware. Second, whenever operation I_1 has a data dependency on I_2, I_2's end time must precede I_1's start time. Third are resource constraints – for instance, one technical limitation of the Vericert back end is that it produces hardware with only a single RAM controller, which gives rise to the constraint that no two memory operations (loads or stores) may be scheduled for the same cycle.

These are all passed to a linear program (LP) solver; we use `lp_solve` [8]. The solver outputs a mapping from instructions to states (clock cycles). We reconstruct from this mapping an RtlPar block. Data-dependent instructions mapped to the same state are placed into the same chain, at the innermost level of the RtlPar block; independent instructions mapped to the same state are placed in different chains in the same parallel group; and instructions mapped to different states are placed in different groups (the outermost list of the RtlPar block).

Figure 5 shows an example of our scheduler in action. In fig. 5a, we see the RtlBlock hyperblock to be scheduled. It contains six predicated operations: two additions, three multiplications, and a predicate assignment. The scheduler analyses the hyperblock and constructs a dependency graph (fig. 5b). Each edge of the graph is annotated with the combinational delay of the operation at its head and an estimate for the delay incurred by the path produced by the edge. For example, every edge that leads to operation ⑧ **E**(`goto 10`) is annotated with a delay of 0 because the assignment to the 'state' variable (`state`) is performed immediately and the path delay is assumed to be negligible. These two delays are combined so that the graph can be reasoned about as a flow-graph to find the longest combinational path between two nodes.

```
① [         r2 := r1 + r4;
⑥   p1 =>   p3 := r4 == r2;
②   p1 =>   r1 := r2 + r4;
③   !p1 & !p2 =>
            r3 := r1 * r1;
④   p2 =>   r3 := r1 * r4;
⑦   p2 =>   E(goto 10);
⑤   !p2 =>  r3 := r3 * r3;
⑧           E(goto 10);     ]
```

(a) RtlBlock hyperblock to be scheduled.



(b) Data dependencies in the RtlBlock hyperblock (with transitive dependencies removed).

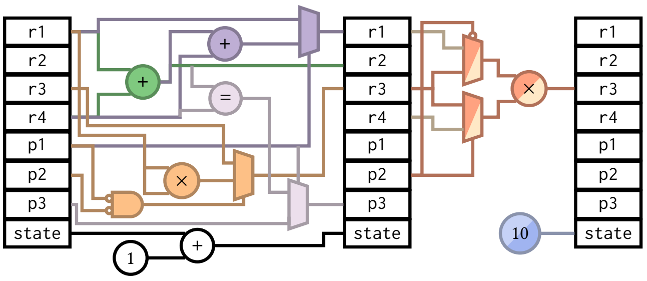
```
③ [[[ !p2 & !p1 =>
                r3 := r1 * r1 ];
①   [           r2 := r1 + r4;
②        p1 => r1 := r2 + r4;
⑥        p1 => p3 := r4 == r2 ]];
⑤  [[ !p2 => r3 := r3 * r3 ];
④   [  p2 => r3 := r1 * r4 ];
⑧   [        E(goto 10); ]]]
```

(c) Scheduled RtlPar hyperblock.

```
state 1:
③ r3 = !p2 & !p1 ? r1 * r1 : r3;
① r2 = r1 + r4;
② r1 = p1 ? r2 + r4 : r1;
⑥ p3 = p1 ? r4 == r2;
   state = 32'd2;

state 2:
⑤ r3 = !p2 ? r3 * r3 : r3;
④ r3 = p2 ? r1 * r4 : r3;
⑧ state = 32'd10;
```

(d) Htl translation of the hyperblock, where instructions are placed into states and are then flattened from the original RtlPar block and translated into a sequence of Verilog blocking assignments.



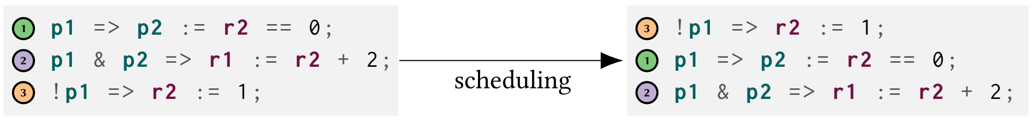
(e) Diagram showing how the operations in the RtlPar hyperblock are scheduled. The circuit is drawn unrolled, so the duplicated registers are actually the same physical register. The colours represent which operations the resources belong to, showing that operations can be chained and shared.

Fig. 5. Example of scheduling a hyperblock.

The scheduler exploits predicates to eliminate dependencies. For example, ③ and ④ appear dependent due to a write-after-write (WAW) conflict on r3, but because their predicates are mutually exclusive, the conflict can be removed from the dependency graph. The scheduler would also remove operations whose predicates are false.

The scheduler transforms the RtlBlock hyperblock into a RtlPar hyperblock (fig. 5c). Even though the addition in ② and the comparison in ⑥ both depend on ①, they can still be placed into the same state because the addition has a short enough combinational delay that two additions can be performed in a single clock cycle. The multiplication in ③ can also be placed into the same state as it does not have any data dependencies with any of the other instructions. The next state has two independent multiplications, ④ and ⑤, that can be scheduled for the same cycle. Finally, the hyperblock is terminated by a control-flow instruction that jumps to state 10. This operation needs to be scheduled after all the other operations, but because it is performed by simply setting the next state of the state machine, this can be done in parallel with the last operation.

Figure 5d shows the corresponding Htl blocks. Htl maps each state to a sequence of Verilog blocking assignments. The translation from RtlPar to Htl first translates each element of the outer list into a separate Htl state. Then, the sequence of blocking assignments is produced by flattening



```
① p1 => p2 := r2 == 0;
② p1 & p2 => r1 := r2 + 2;
③ !p1 => r2 := 1;
```

```
③ !p1 => r2 := 1;
① p1 => p2 := r2 == 0;
② p1 & p2 => r1 := r2 + 2;
```

Fig. 6. An example schedule. It is valid to move ③ before ① and ② because despite the appearance of data dependencies on `r2`, ③ is in fact independent because its guard is mutually exclusive with the other two.

Table 1. First attempt: basic symbolic execution.

	Pre-scheduling symbolic state	**Post-scheduling symbolic state**
`r1`	$\texttt{if } (\texttt{p1}^0 \wedge (\texttt{if p1}^0 \texttt{ then r2}^0 == 0 \texttt{ else p2}^0))$ $\texttt{then r2}^0 + 2$ $\texttt{else r1}^0$	$\texttt{if } \left(\texttt{p1}^0 \wedge \left(\texttt{if p1}^0 \texttt{ then } \left(\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else r2}^0\right) == 0 \texttt{ else p2}^0\right)\right)$ $\texttt{then } (\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else r2}^0) + 2$ $\texttt{else r1}^0$
`r2`	$\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else r2}^0$	$\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else r2}^0$
`p2`	$\texttt{if p1}^0 \texttt{ then r2}^0 == 0 \texttt{ else p2}^0$	$\texttt{if p1}^0$ $\texttt{then } (\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else r2}^0) == 0$ $\texttt{else p2}^0$

the remaining inner nested lists. Verilog's blocking assignment has a sequential semantics, meaning it behaves like the RtlPar block.

Figure 5e shows how the resultant hardware executes. In particular, the two multiplications ④ and ⑤ could be allocated to the same resources since they never execute at the same time.

6 VALIDATION OF HYPERBLOCK SCHEDULING

Although the scheduling algorithm itself is complex with many heuristics, it is quite simple to check each specific schedule. To do so, we follow Tristan and Leroy [43] and symbolically execute each block before and after scheduling, then compare the two obtained symbolic states for equivalence. The main difference from Tristan and Leroy's approach is that our paths are represented by predicates instead of by explicit branches. As a result, several non-obvious design decisions need to be made so that the validation process is tractable in the presence of hyperblocks. In what follows, we explain these decisions informally with the aid of the example shown in fig. 6. We then present our validator more precisely in section 6.5.

6.1 First Attempt: Basic Symbolic Execution

The most natural way to extend Tristan and Leroy's approach is to treat predicates in the same way as registers. Symbolic execution then yields a symbolic state that assigns to each register and predicate an expression that is in terms of the initial values of the registers and predicates. Applying this approach to the example in fig. 6 produces the two symbolic states shown in table 1. Note that we write $\texttt{r2}^0$ for the initial value of `r2`, and so on.

The pre- and post-scheduling expressions for `r2` are syntactically equal, but reasoning about the equivalence of the two expressions for `p2` is more involved: our validator needs to understand that `if` $\neg$`p1`^{0} `then` 1 `else` `r2`^{0} is equivalent to `r2`^{0} in any context where `p1`^{0} is true. Such reasoning could be performed by an SMT solver, encoding each arithmetic operator as an uninterpreted function, but formalising an SMT solver involves a lot of additional proof, and would be slow at run time.

6.2 Second Attempt: Using Value Summaries

Instead we would prefer to rely on a SAT solver, as it is easier to formalise and verify. A SAT solver can handle Boolean reasoning nicely, but cannot reason about arithmetic. So, to allow the use of a SAT solver, we rewrite each expression into a normal form where all the if-expressions are pulled to the top level, e.g. replacing (`if` $\neg$`p1`^{0} `then` 1 `else` `r2`^{0}) == 0 with `if` $\neg$`p1`^{0} `then` 1 == 0 `else` `r2`^{0} == 0. On our worked example (fig. 6), this results in the symbolic states shown in table 2.

Note that we treat register expressions and predicate expressions slightly differently. For register expressions, we combine all the if-expressions into a single multi-way conditional, which we write using "cases" notation. We call these expressions *value summaries* after Sen et al. [38], who used the same data structure for a different purpose (namely, making symbolic execution more efficient).

For predicate expressions, we do not need value summaries, because all of the `if` e_1 `then` e_2 `else` e_3 operations that appear in the predicate expressions become purely Boolean (rather than a mix of integers and Booleans), and hence can be expanded to $(e_1 \longrightarrow e_2) \land (\neg e_1 \longrightarrow e_3)$, as we have done in table 2. These predicate expressions can then be straightforwardly translated into propositional formulas that can be reasoned about using a SAT solver. For example, the generated query for checking the equivalence between the expressions assigned to `p2` looks like the following:

$$((x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}) \land (\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0})) \leftrightarrow S_\psi \tag{1}$$

where S_ψ abbreviates $((x_{\mathtt{p1}^0} \longrightarrow ((\neg x_{\mathtt{p1}^0} \longrightarrow x_{1==0}) \land (\neg\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}))) \land (\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0}))$. In the encoding, each SAT variable x_e encodes the truth value of the expression e in the formula.

We can also generate SAT queries to check the equivalence of the register expressions. This involves issuing multiple queries to the SAT solver: if an expression appears in both the pre- and post-scheduling value summaries, then we generate a query to check that their guards are equivalent, and if an expression only appears in one of the value summaries, then its guard should be equivalent to *false*. For instance, to check `r1` we generate the following three SAT queries:

$$\begin{aligned} \mathit{false} &\leftrightarrow x_{\mathtt{p1}^0} \land S_\psi \land \neg x_{\mathtt{p1}^0} \\ (x_{\mathtt{p1}^0} \land ((x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}) \land (\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0}))) &\leftrightarrow (x_{\mathtt{p1}^0} \land S_\psi \land x_{\mathtt{p1}^0}) \\ \neg(x_{\mathtt{p1}^0} \land (((x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}) \land (\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0})))) &\leftrightarrow \neg(x_{\mathtt{p1}^0} \land S_\psi) \end{aligned} \tag{2}$$

However, in constructing all these SAT queries, we have assumed that a Boolean value can be assigned to each of the atoms in a formula. This might not actually be the case – for instance, `r1/r2` is not evaluable if `r2` is zero, and `r1 == r2` is not evaluable if either `r1` or `r2` is an invalid pointer. So, to compare the two expressions for `p2`, we actually need to use *3-valued logic*. That means all the SAT variables in (1) and (2) actually need to be *pairs* of binary variables, and the $\land$ and $\lor$ operations need to be 3-valued analogues of conjunction and disjunction. This is a problem because we have seen already that these formulas, particularly those for comparing register expressions, become quite large even for toy examples. Indeed, when we tried this 3-valued approach on the test cases in our evaluation (section 8), most ran out of memory during validation.

Hence, in the next subsection we describe how we manage to avoid 3-valued logic where possible.

Table 2. Second attempt: using value summaries.

	Pre-scheduling symbolic state	Post-sched. symbolic state
r1	$\begin{cases} \texttt{r2}^0 + 2, & \text{if } \texttt{p1}^0 \land (\texttt{p1}^0 \to \texttt{r2}^0 == 0) \land (\neg\texttt{p1}^0 \to \texttt{p2}^0) \\ \texttt{r1}^0, & \text{if } \neg(\texttt{p1}^0 \land (\texttt{p1}^0 \to \texttt{r2}^0 == 0) \land (\neg\texttt{p1}^0 \to \texttt{p2}^0)) \end{cases}$	$\begin{cases} 1 + 2, & \text{if } \texttt{p1}^0 \land \psi \land \neg\texttt{p1}^0 \\ \texttt{r2}^0 + 2, & \text{if } \texttt{p1}^0 \land \psi \land \texttt{p1}^0 \\ \texttt{r1}^0, & \text{if } \neg(\texttt{p1}^0 \land \psi) \end{cases}$
r2	$\begin{cases} 1, & \text{if } \neg\texttt{p1}^0 \\ \texttt{r2}^0, & \text{if } \texttt{p1}^0 \end{cases}$	$\begin{cases} 1, & \text{if } \neg\texttt{p1}^0 \\ \texttt{r2}^0, & \text{if } \texttt{p1}^0 \end{cases}$
p2	$(\texttt{p1}^0 \to \texttt{r2}^0 == 0) \land (\neg\texttt{p1}^0 \to \texttt{p2}^0)$	ψ

where ψ abbreviates $(\texttt{p1}^0 \to ((\neg\texttt{p1}^0 \to 1 == 0) \land (\neg\neg\texttt{p1}^0 \to \texttt{r2}^0 == 0))) \land (\neg\texttt{p1}^0 \to \texttt{p2}^0)$

6.3 Third Attempt: Using Value Summaries and Final-State Guards

Although it is not the case in our worked example, it turns out that when comparing the predicate expressions that arise in realistic examples, syntactic equality or near-equality usually suffices. This means that we only need to resort to solving 3-valued SAT queries as an occasional fallback, so its performance impact is limited in practice.

Where syntactic methods usually do *not* suffice is for comparing the guards of register expressions. However, here we can actually avoid the need for 3-valued logic altogether. We observe that the guards in the r1 expressions are simply copied from the expressions for p2 (which we abbreviated as ψ in table 2), so rather than writing out the full expressions in the guards, we can write $\texttt{p2}^{\mathrm{f}}$ as a shorthand (the 'f' clarifies that it is referring to the *final* value of p2).

To make it possible to refer to final values, we need to ensure that once p2 has been assigned or used, it is never overwritten. We can achieve this by enforcing SSA form for predicate assignments. That does not impose any restrictions on the Vericert user because predicates are only introduced by internal compiler transformations. SSA form is not needed for register assignments.

The resultant symbolic states are shown in table 3. It can immediately be seen that the expressions have become much shorter. Indeed, the three queries for validating r1 become:

$$\begin{aligned} \mathit{false} &\leftrightarrow x_{\texttt{p1}^{\mathrm{f}}} \land x_{\texttt{p2}^{\mathrm{f}}} \land \neg x_{\texttt{p1}^{\mathrm{f}}} \\ x_{\texttt{p1}^{\mathrm{f}}} \land x_{\texttt{p2}^{\mathrm{f}}} &\leftrightarrow x_{\texttt{p1}^{\mathrm{f}}} \land x_{\texttt{p2}^{\mathrm{f}}} \land x_{\texttt{p1}^{\mathrm{f}}} \\ \neg(x_{\texttt{p1}^{\mathrm{f}}} \land x_{\texttt{p2}^{\mathrm{f}}}) &\leftrightarrow \neg(x_{\texttt{p1}^{\mathrm{f}}} \land x_{\texttt{p2}^{\mathrm{f}}}) \end{aligned} \tag{3}$$

What is less obvious is that we no longer need 3-valued logic either. This is because:

- We can assume that all predicate expressions in the pre-scheduling symbolic state are evaluable, because if any were not, the input program would fail at run time and we do not need to prove anything about our scheduler.
- We have already proven, either using syntactic comparison or 3-valued logic, that the predicate expressions in the post-scheduling symbolic state are equivalent to those in the pre-scheduling state, which means that they too must be evaluable.
- The guards in the register expressions only refer to these expressions – they cannot include unsafe expressions like division or pointer comparison – and so they must also be evaluable.
- It therefore suffices to use 2-valued logic to compare the guards.

Table 3. Third attempt: using value summaries and final values in guards.

	Pre-scheduling symbolic state	**Post-scheduling symbolic state**
r1	$\begin{cases} r2^0 + 2, & \text{if } p1^f \land p2^f \\ r1^0, & \text{if } \neg(p1^f \land p2^f) \end{cases}$	$\begin{cases} 1 + 2, & \text{if } p1^f \land p2^f \land \neg p1^f \\ r2^0 + 2, & \text{if } p1^f \land p2^f \land p1^f \\ r1^0, & \text{if } \neg(p1^f \land p2^f) \end{cases}$
r2	$\begin{cases} 1, & \text{if } \neg p1^f \\ r2^0, & \text{if } p1^f \end{cases}$	$\begin{cases} 1, & \text{if } \neg p1^f \\ r2^0, & \text{if } p1^f \end{cases}$
p2	$(p1^0 \longrightarrow r2^0 == 0) \land (\neg p1^0 \longrightarrow p2^0)$	ψ

where ψ abbreviates $(p1^0 \longrightarrow ((\neg p1^0 \longrightarrow 1 == 0) \land (\neg\neg p1^0 \longrightarrow r2^0 == 0))) \land (\neg p1^0 \longrightarrow p2^0)$

6.4 Handling Overwritten Expressions

There is an additional subtlety that needs to be handled: the possibility that the scheduler introduces undefined behaviour. Consider the following example, due to Tristan and Leroy [43].

```
r3 := r2 + 4;
```

$\xrightarrow{\text{scheduling}}$

```
r3 := 5 / r1;
r3 := r2 + 4;
```

Symbolic execution yields identical pre- and post-scheduling results, namely $r3 \mapsto r2^0 + 4$. Despite this, the schedule is invalid because the post-scheduling block only executes correctly when `r1` is nonzero. To detect and forbid such cases, we follow Tristan and Leroy and keep track of all the expressions that are evaluated into a register or memory location. We shall call this the *encountered expression set.*[3] For example, the encountered expressions of the pre-scheduling block above includes only $r2^0 + 4$, but the post-scheduling block's also includes $5/r1^0$. Because the encountered set has grown, we deem the schedule invalid.

In order to use Tristan and Leroy's approach with hyperblocks, it needs extending to handle predicated instructions. The obvious way to do this is to generate the encountered expressions for each predicated instruction in the same way that we perform symbolic execution, which is essentially to treat `p => r := e` as if it is the non-predicated instruction `r := p ? e : r`. However, this approach leads to too many unnecessary constraints being imposed on the scheduler, leaving it unable to reorder some instructions that have only benign WAW dependencies. To see this, consider the following example.

```
p1 => r1 := 1;
!p1 => r1 := 2;
```

$\xrightarrow{\text{scheduling}}$

```
!p1 => r1 := 2;
p1 => r1 := 1;
```

When performing symbolic execution on the pre-scheduling block, we encounter the pair of expressions $\texttt{if } p1^f \texttt{ then } 1 \texttt{ else } r1^0$ and $\texttt{if } \neg p1^f \texttt{ then } 2 \texttt{ else if } p1^f \texttt{ then } 1 \texttt{ else } r1^0$, but on the post-scheduling block we encounter $\texttt{if } \neg p1^f \texttt{ then } 2 \texttt{ else } r1^0$ and $\texttt{if } p1^f \texttt{ then } 1 \texttt{ else if } \neg p1^f \texttt{ then } 2 \texttt{ else } r1^0$; these two pairs are not equivalent, so this (correct) schedule cannot be validated.

Instead, for the purposes of calculating encountered expressions, our approach is to treat `p => r := e` as the instruction `r := p ? e : •`, where • is a dummy expression representing the absence of an assignment. The set of encountered expressions is now the same for both pre- and post-scheduling: $\{\texttt{if } p1^f \texttt{ then } 1 \texttt{ else } \bullet, \texttt{if } \neg p1^f \texttt{ then } 2 \texttt{ else } \bullet\}$, so validation can be completed.

[3]Tristan and Leroy simply called them 'constraints'.

arithmetic expressions:	$\mathbb{A} ::= \mathbb{r}^0$	(initial value of register)
	$\mid \mathbb{M}[\mathbb{A}]$	(load from memory)
	$\mid \mathbb{A}\ \mathbb{a}\ \mathbb{A}$	(binary arithmetic operation)
memory expressions:	$\mathbb{M} ::= \mathtt{Mem}^0$	(initial contents of memory)
	$\mid \mathbb{M}[\mathbb{A}\ \texttt{->}\ \mathbb{A}]$	(updated memory)
predicate expressions:	$\mathbb{B} ::= \mathbb{p}^0 \mid \neg\mathbb{p}^0$	(initial value of predicate)
	$\mid \mathbb{A}\ \mathbb{c}\ \mathbb{A} \mid \neg(\mathbb{A}\ \mathbb{c}\ \mathbb{A})$	(binary conditional operation)
	$\mid \mathit{true} \mid \mathit{false} \mid \mathbb{B} \wedge \mathbb{B}$	(true, false, and, or)
	$\mid \mathbb{B} \vee \mathbb{B}$	
value summaries:	$\mathcal{S}(t) \ = \mathcal{P}(\mathbb{G} \times t)$	(select an element of t according to which predicate holds)
symbolic states:	$\sigma_\mathrm{R} \in \Sigma_\mathrm{R} \ = \mathbb{r} \to \mathcal{S}(\mathbb{A})$	(expressions for registers)
	$\sigma_\mathrm{P} \in \Sigma_\mathrm{P} \ = \mathbb{p} \to \mathbb{B}$	(expressions for predicates)
	$\sigma_\mathrm{M} \in \Sigma_\mathrm{M} \ = \mathcal{S}(\mathbb{M})$	(contents of memory)
	$\sigma_\mathrm{E} \in \Sigma_\mathrm{E} \ = \mathcal{S}(\mathbb{I}^?_\mathrm{cf})$	(instruction to exit block)
	$\sigma_\mathrm{C} \in \Sigma_\mathrm{C} \ = \mathcal{P}(\mathcal{S}(\mathbb{A} + \mathbb{M}))$	(set of encountered expressions)

Fig. 7. Syntax of symbolic states.

6.5 Formalising the Symbolic State and Symbolic Execution

The previous sections gave an informal overview of the structure of the symbolic state and the validation algorithm. This section will give formal definitions of these concepts.

Symbolic states. Figure 7 defines the symbolic states that symbolic execution produces. Several components make use of value summaries (as explained in section 6.2), so we define the value summary $\mathcal{S}(t)$ as a set of terms of type t, each paired with a Boolean guard of type $\mathbb{G}$. Henceforth, we shall sometimes write value summaries explicitly as a set of (guard, value) pairs.

A symbolic state σ is made up of five components, the main three being: a register map σ_R that assigns an arithmetic expression (as a value summary) to each register, a predicate map σ_P that assigns a Boolean expression to each predicate, and an expression σ_M for the contents of memory (again as a value summary). We also need symbolic execution to track how control exits the block (to make sure that it does so in the same way after scheduling), so σ_E stores a value summary that evaluates to the instruction that is executed to exit the block (or to '`None`' if the block hasn't finished yet). Finally, σ_C tracks the set of encountered expressions, as motivated in section 6.4. The symbolic states presented by Tristan and Leroy were structured in a similar manner, requiring σ_R, σ_M and σ_C, but not needing value summaries to represent expressions. They also did not need σ_E because basic blocks could not represent arbitrary exits.

Constructing symbolic states. The expressions are constructed using a function which updates the symbolic expressions assigned for each resource. A core function used to update value summaries is the coalescing union operator $\uplus_q$ [38], which conjoins $\neg q$ to each guard in its left operand and q to each guard in its right operand:

$$\begin{aligned} &\uplus_q \in \mathcal{S}(\mathbb{X}) \to \mathcal{S}(\mathbb{X}) \to \mathcal{S}(\mathbb{X}) \\ &X_1 \uplus_q X_2 \triangleq \{(\neg q \wedge G, v) \mid (G, v) \in X_1\} \cup \{(q \wedge G, v) \mid (G, v) \in X_2\} \end{aligned} \tag{4}$$

$$
\begin{array}{ll}
\alpha\ (G \texttt{ => } \texttt{r := r1 + r2})\ (q, (\sigma_{\mathrm{R}}, \sigma_{\mathrm{P}}, \sigma_{\mathrm{M}}, \sigma_{\mathrm{E}}, \sigma_{\mathrm{C}})) & \text{(arithmetic operation)} \\
\quad \triangleq \texttt{let } \phi = \{ (\mathit{true}, (+)) \} \texttt{ <*> } \sigma_{\mathrm{R}}[\texttt{r1}] \texttt{ <*> } \sigma_{\mathrm{R}}[\texttt{r2}] \texttt{ in} & \\
\qquad \texttt{let } \sigma'_{\mathrm{R}} = \sigma_{\mathrm{R}}\big[\texttt{r} \mapsto (\sigma_{\mathrm{R}}[\texttt{r}] \uplus_{G \wedge q} \phi)\big] \texttt{ in} & \\
\qquad \texttt{let } \sigma'_{\mathrm{C}} = \sigma_{\mathrm{C}} \cup \{ \{ (\mathit{true}, \bullet) \} \uplus_{G \wedge q} \phi \} \texttt{ in} & \\
\qquad (q, (\sigma'_{\mathrm{R}}, \sigma_{\mathrm{P}}, \sigma_{\mathrm{M}}, \sigma_{\mathrm{E}}, \sigma'_{\mathrm{C}})) & \\
\alpha\ (G \texttt{ => } \mathsf{E}(I_{\mathrm{cf}}))\ (q, (\sigma_{\mathrm{R}}, \sigma_{\mathrm{P}}, \sigma_{\mathrm{M}}, \sigma_{\mathrm{E}}, \sigma_{\mathrm{C}})) & \text{(exit instruction)} \\
\quad \triangleq \texttt{let } \sigma'_{\mathrm{E}} = \sigma_{\mathrm{E}} \uplus_{G \wedge q} \{ (\mathit{true}, \mathsf{Some}(I_{\mathrm{cf}})) \} \texttt{ in} & \\
\qquad \big(q \wedge \neg G, \big(\sigma_{\mathrm{R}}, \sigma_{\mathrm{P}}, \sigma_{\mathrm{M}}, \sigma'_{\mathrm{E}}, \sigma_{\mathrm{C}}\big)\big) & \\
\alpha\ (G \texttt{ => } \texttt{p := r1 == r2})\ (q, (\sigma_{\mathrm{R}}, \sigma_{\mathrm{P}}, \sigma_{\mathrm{M}}, \sigma_{\mathrm{E}}, \sigma_{\mathrm{C}})) & \text{(predicate assignment)} \\
\quad \triangleq \texttt{let } \phi = \wedge(\{ (G \wedge q, (==)) \} \texttt{ <*> } \sigma_{\mathrm{R}}[\texttt{r1}] \texttt{ <*> } \sigma_{\mathrm{R}}[\texttt{r2}]) \texttt{ in} & \\
\qquad \texttt{let } \sigma'_{\mathrm{P}} = \sigma_{\mathrm{P}}[\texttt{p} \mapsto (\phi \wedge (\neg(G \wedge q) \rightarrow \sigma_{\mathrm{P}}[\texttt{p}]))] \texttt{ in} & \\
\qquad (q, (\sigma_{\mathrm{R}}, \sigma'_{\mathrm{P}}, \sigma_{\mathrm{M}}, \sigma_{\mathrm{E}}, \sigma_{\mathrm{C}})) &
\end{array}
$$

Fig. 8. Symbolic execution of selected instructions

To turn a value summary back into a Boolean formula, we use the following operation, where g expands gates into predicate expressions:

$$
\begin{aligned}
&\wedge \in \mathcal{S}(\mathbb{B}) \rightarrow \mathbb{B} \\
&\wedge\{ (G_1, B_1), \ldots, (G_n, B_n) \} \triangleq (g(G_1) \rightarrow B_1) \wedge \cdots \wedge (g(G_n) \rightarrow B_n)
\end{aligned} \tag{5}
$$

It is also useful to have an applicative interface for value summaries, so that when we have value summaries of functions and of inputs, we can obtain a value summary of outputs:

$$
\begin{aligned}
&\texttt{<*>} \in \mathcal{S}(\mathbb{X} \rightarrow \mathbb{Y}) \rightarrow \mathcal{S}(\mathbb{X}) \rightarrow \mathcal{S}(\mathbb{Y}) \\
&F \texttt{ <*> } X \triangleq \{ (G \wedge G', f(x)) \mid (G, f) \in F, (G', x) \in X \}
\end{aligned} \tag{6}
$$

Following Sen et al. [38], we simplify value summaries as they are built up, so as to keep their size from exploding: coalescing two elements (G, v) and (G', v') where $v = v'$ into a single element $(G \vee G', v)$, and removing elements (G, v) whenever $G \leftrightarrow \mathit{false}$.

Symbolic execution. The symbolic execution of instruction I is performed by the α function. It takes the current symbolic state σ and produces an updated one. It also takes an 'enabled' predicate q, which is conjoined with the current instruction's guard; it ensures that after an exit instruction is taken, any subsequent instructions are nullified. So, whenever an exit instruction is encountered, the enabled predicate is conjoined with the negation of the exit instruction's guard.

In fig. 8, we show three important cases of α: symbolically executing an arithmetic operation, an exit instruction, and a predicate assignment. To symbolically execute a whole hyperblock (denoted $\cdot^{\#}$), we run α on each instruction in turn, threading the symbolic state through, starting from the empty symbolic state (denoted $\varnothing$):

$$
[i_1; i_2; \ldots; i_n]^{\#} \triangleq \alpha\ i_n\ (\ldots (\alpha\ i_2\ (\alpha\ i_1\ (\mathit{true}, \varnothing))) \ldots) \tag{7}
$$

Comparing symbolic states. After symbolically executing the RtlBlock and RtlPar blocks, we obtain two symbolic states, σ and σ'. We wish to show that σ' is a *symbolic refinement* of σ (written $\sigma \gtrsim \sigma'$), and we do so by component-wise comparison, as shown in eq. (8) and explained below.

$$
\frac{\sigma_{\mathrm{R}} \approx \sigma'_{\mathrm{R}} \qquad \sigma_{\mathrm{P}} = \sigma'_{\mathrm{P}} \vee \sigma_{\mathrm{P}} \approx_{3\mathrm{v}} \sigma'_{\mathrm{P}} \qquad \sigma_{\mathrm{M}} \approx \sigma'_{\mathrm{M}} \qquad \sigma_{\mathrm{E}} \approx \sigma'_{\mathrm{E}} \qquad \sigma_{\mathrm{C}} \gtrsim \sigma'_{\mathrm{C}}}{\sigma \gtrsim \sigma'} \tag{8}
$$

The core comparison operation that we rely upon is between two value summaries, written $\approx$. Whenever a value appears in both value summaries, we check that its guards are equivalent (via a SAT query), and whenever a value appears in just one value summary, we check that its guard is equivalent to *false* (again via SAT query). This approach suffices for the register maps ($\sigma_R \approx \sigma'_R$), the memory maps ($\sigma_M \approx \sigma'_M$), and the exit expressions ($\sigma_E \approx \sigma'_E$). For the predicate maps, we first attempt to show syntactic equality ($\sigma_P = \sigma'_P$). If this fails, we fall back to using a slow but reliable equivalence check with a 3-valued solver ($\sigma_P \approx_{3v} \sigma'_P$). Finally, for the encountered expressions sets, we write $\sigma_C \gtrsim \sigma'_C$ to mean that every expression in σ'_C has an equivalent in σ_C.

6.6 Defining a Verified Scheduler

We can now define a verified scheduler using the standard translation validation approach:

$$\begin{aligned} \texttt{scheduleAndVerify}\, H \triangleq\ & \texttt{let}\ H_{\text{par}} = \texttt{schedule}\ H\ \texttt{in} \\ & \texttt{if}\ H^{\#} \gtrsim H^{\#}_{\text{par}}\ \texttt{then}\ \texttt{Some}(H_{\text{par}})\ \texttt{else}\ \texttt{None} \end{aligned} \tag{9}$$

7 PROVING THE VALIDATOR CORRECT

In order to prove our validator correct, we need to prove that whenever our validator deems $H^{\#}_{\text{par}}$ to be a symbolic refinement of $H^{\#}$, there is indeed a forward simulation from H to H_{par} (defined more precisely in definition 7.1); that is:

$$H^{\#} \gtrsim H^{\#}_{\text{par}} \implies H \rightsquigarrow H_{\text{par}} \tag{10}$$

The natural way to prove $H \rightsquigarrow H_{\text{par}}$ is to follow Tristan and Leroy [43] by constructing the chain $H \rightsquigarrow H^{\#} \rightsquigarrow H^{\#}_{\text{par}} \rightsquigarrow H_{\text{par}}$. In order to do this, we need to be able to talk about forward simulations that involve not just programs (H and H_{par}) but also symbolic states ($H^{\#}$ and $H^{\#}_{\text{par}}$). Thus we need a semantics not just for blocks (cf. fig. 3) but also for symbolic states.

7.1 A Semantics for Symbolic States

We need a function that takes a symbolic state σ and applies it to an initial concrete state Γ. The output is the concrete state Γ', together with the control-flow instruction I_{cf} that is executed to exit the block. The function is written as $\Gamma \vdash \sigma \Downarrow (\Gamma', I_{\text{cf}})$, and is defined in fig. 9. It works as follows:

- The entry point is the SemState rule. This rule has six antecedents. The first constructs a Boolean value for each predicate in the final state. The second constructs a value for each register in the final state, consulting Γ'_P to get the final values of predicates when evaluating value summaries (cf. section 6.3). The third constructs the final contents of memory. The fourth determines the control-flow instruction for exiting the block and the fifth expands Γ. The sixth does not calculate a component of the final state; instead, its purpose is to prevent the final state being calculated at all if any of the encountered expressions (cf. section 6.4) are unevaluable.
- The $\Downarrow_{\mathbb{A}}$ rules (RegBase, Load, and Op) map register expressions to concrete values (integer, float, pointer, or 'undefined'). In the Op rule, we write $\downarrow$ to indicate the existing CompCert evaluation semantics for the arithmetic operation op_a, which may need to consult Γ_{Env} to handle operands that are relative to the stack pointer or a global variable.
- The $\Downarrow_{\mathbb{M}}$ rules map memory expressions to concrete values.
- The ArithEmpty and MemEmpty rules map the dummy expression $\bullet$ to an arbitrary arithmetic value (1), or to an arbitrary memory (the initial memory). This is so we can check that all encountered expressions are evaluable; their actual values are immaterial.

$$\textsc{RegBase}\ \frac{}{\Gamma \vdash r^0 \Downarrow_{\mathbb{A}} \Gamma_{\mathrm{R}}[r]} \qquad \textsc{PredBase}\ \frac{}{\Gamma \vdash p^0 \Downarrow_{\mathbb{B}} \Gamma_{\mathrm{P}}[p]} \qquad \textsc{MemBase}\ \frac{}{\Gamma \vdash \mathtt{Mem}^0 \Downarrow_{\mathbb{M}} \Gamma_{\mathrm{M}}} \qquad \textsc{Option}\ \frac{}{\Gamma \vdash \mathtt{Some}(a) \Downarrow_{\mathbb{I}^?_{\mathrm{cf}}} a}$$

$$\textsc{Op}\ \frac{\Gamma \vdash e_1 \Downarrow_{\mathbb{A}} v_1 \qquad \Gamma \vdash e_2 \Downarrow_{\mathbb{A}} v_2 \qquad \Gamma \vdash v_1\ op_{\mathrm{a}}\ v_2 \downarrow v}{\Gamma \vdash e_1\ op_{\mathrm{a}}\ e_2 \Downarrow_{\mathbb{A}} v} \qquad \textsc{Pred}\ \frac{\Gamma \vdash e_1 \Downarrow_{\mathbb{A}} v_1 \qquad \Gamma \vdash e_2 \Downarrow_{\mathbb{A}} v_2 \qquad \Gamma \vdash v_1\ op_{\mathrm{c}}\ v_2 \downarrow \mathtt{Some}(b)}{\Gamma \vdash e_1\ op_{\mathrm{c}}\ e_2 \Downarrow_{\mathbb{B}} b}$$

$$\textsc{Store}\ \frac{\Gamma \vdash e_1 \Downarrow_{\mathbb{M}} m \qquad \Gamma \vdash e_2 \Downarrow_{\mathbb{A}} i \qquad \Gamma \vdash e_3 \Downarrow_{\mathbb{A}} v}{\Gamma \vdash e_1[e_2 \mathtt{->} e_3] \Downarrow_{\mathbb{M}} m[i \mapsto v]} \qquad \textsc{Load}\ \frac{\Gamma \vdash e_1 \Downarrow_{\mathbb{M}} m \qquad \Gamma \vdash e_2 \Downarrow_{\mathbb{A}} i}{\Gamma \vdash e_1[e_2] \Downarrow_{\mathbb{A}} m[i]} \qquad \textsc{PredAndTrue}\ \frac{\Gamma \vdash B_1 \Downarrow_{\mathbb{B}} \mathit{true} \qquad \Gamma \vdash B_2 \Downarrow_{\mathbb{B}} \mathit{true}}{\Gamma \vdash B_1 \wedge B_2 \Downarrow_{\mathbb{B}} \mathit{true}}$$

$$\textsc{PredAndFalse1}\ \frac{\Gamma \vdash B_1 \Downarrow_{\mathbb{B}} \mathit{false}}{\Gamma \vdash B_1 \wedge B_2 \Downarrow_{\mathbb{B}} \mathit{false}} \qquad \textsc{PredAndFalse2}\ \frac{\Gamma \vdash B_2 \Downarrow_{\mathbb{B}} \mathit{false}}{\Gamma \vdash B_1 \wedge B_2 \Downarrow_{\mathbb{B}} \mathit{false}} \qquad \textsc{PredOrTrue1}\ \frac{\Gamma \vdash B_1 \Downarrow_{\mathbb{B}} \mathit{true}}{\Gamma \vdash B_1 \vee B_2 \Downarrow_{\mathbb{B}} \mathit{true}} \qquad \textsc{PredOrTrue2}\ \frac{\Gamma \vdash B_2 \Downarrow_{\mathbb{B}} \mathit{true}}{\Gamma \vdash B_1 \vee B_2 \Downarrow_{\mathbb{B}} \mathit{true}}$$

$$\textsc{PredOrFalse}\ \frac{\Gamma \vdash B_1 \Downarrow_{\mathbb{B}} \mathit{false} \qquad \Gamma \vdash B_2 \Downarrow_{\mathbb{B}} \mathit{false}}{\Gamma \vdash B_1 \vee B_2 \Downarrow_{\mathbb{B}} \mathit{false}} \qquad \textsc{ArithEmpty}\ \frac{}{\Gamma \vdash \bullet \Downarrow_{\mathbb{A}} 1} \qquad \textsc{MemEmpty}\ \frac{}{\Gamma \vdash \bullet \Downarrow_{\mathbb{M}} \Gamma_{\mathrm{M}}}$$

$$\textsc{Sum1}\ \frac{\Gamma \vdash e \Downarrow_t v}{\Gamma \vdash e \Downarrow_{t+u} v} \qquad \textsc{Sum2}\ \frac{\Gamma \vdash e \Downarrow_u v}{\Gamma \vdash e \Downarrow_{t+u} v} \qquad \textsc{PredExpr}\ \frac{(G, e) \in s \qquad P^{\mathrm{f}} \vdash G \downarrow \mathit{true} \qquad \Gamma \vdash e \Downarrow_t v}{P^{\mathrm{f}}, \Gamma \vdash s \Downarrow_{\mathcal{S}(t)} v}$$

$$\textsc{SemState}\ \frac{\begin{array}{c} \forall x.\, \Gamma \vdash \sigma_{\mathrm{P}}[x] \Downarrow_{\mathbb{B}} \Gamma'_{\mathrm{P}}[x] \\ \forall x.\, \Gamma'_{\mathrm{P}}, \Gamma \vdash \sigma_{\mathrm{R}}[x] \Downarrow_{\mathcal{S}(\mathbb{A})} \Gamma'_{\mathrm{R}}[x] \\ \Gamma'_{\mathrm{P}}, \Gamma \vdash \sigma_{\mathrm{M}} \Downarrow_{\mathcal{S}(\mathbb{M})} \Gamma'_{\mathrm{M}} \\ \Gamma'_{\mathrm{P}}, \Gamma \vdash \sigma_{\mathrm{E}} \Downarrow_{\mathcal{S}(\mathbb{I}^?_{\mathrm{cf}})} I_{\mathrm{cf}} \\ \Gamma = (\Gamma_{\mathrm{Env}}, \Gamma_{\mathrm{R}}, \Gamma_{\mathrm{P}}, \Gamma_{\mathrm{M}}) \\ \forall x \in \sigma_{\mathrm{C}}.\, \exists v.\, \Gamma'_{\mathrm{P}}, \Gamma \vdash x \Downarrow_{\mathcal{S}(\mathbb{A}+\mathbb{M})} v \end{array}}{\Gamma \vdash \sigma \Downarrow ((\Gamma_{\mathrm{Env}}, \Gamma'_{\mathrm{R}}, \Gamma'_{\mathrm{P}}, \Gamma'_{\mathrm{M}}), I_{\mathrm{cf}})}$$

Fig. 9. Semantics of symbolic states.

- The $\Downarrow_{\mathbb{B}}$ rules map predicate expressions to Boolean values (*true* and *false*). In the Pred rule, evaluating $v_1\ op_{\mathrm{c}}\ v_2$ returns an option type because v_1 or v_2 might be an invalid pointer.
- The rules for $\wedge$ and $\vee$ are designed to produce a lazy semantics. This is necessitated by the fact that they originate from if-statements in the source program, which must be evaluated lazily. In particular, if B_1 evaluates to *false* and B_2 is unevaluable, then we need $B_1 \wedge B_2$ to evaluate to *false*, not to be unevaluable.
- The PredExpr rule is for evaluating a value summary s of type $\mathcal{S}(t)$. It finds an entry (G, e) for which the guard G evaluates to true in the final state (P^{f}), and then evaluates e at type t. Value summaries are constructed so that the guards are exhaustive and mutually exclusive, so there will always be exactly one such entry.

7.2 Establishing the Chain of Simulations

Now that we have a semantics for symbolic states, we can define the required forward simulation relation, $a \rightsquigarrow b$, where a and b are both blocks, or both symbolic states, or one of each.

Definition 7.1 (Forward lock-step simulation diagram). For every execution of a, there exists an execution of b that, when starting from a matching initial state, results in a matching final state.

$$a \rightsquigarrow b \;\triangleq\; \forall \Gamma_1, \Gamma_1', \Gamma_2, I_{\mathrm{cf}}.\, \Gamma_1 \vdash a \Downarrow (\Gamma_1', I_{\mathrm{cf}}) \wedge \Gamma_1 \sim \Gamma_2 \implies \exists \Gamma_2'.\, \Gamma_2 \vdash b \Downarrow (\Gamma_2', I_{\mathrm{cf}}) \wedge \Gamma_1' \sim \Gamma_2' \qquad (11)$$

It remains to construct the chain $H \rightsquigarrow H^{\#} \rightsquigarrow H^{\#}_{\mathrm{par}} \rightsquigarrow H_{\mathrm{par}}$. The three steps of that chain are captured by the following three lemmas.

Lemma 7.2 (Soundness of symbolic execution). *For every execution of a block H, there exists an equivalent execution of its symbolic state $H^{\#}$. That is: for all H, we have $H \rightsquigarrow H^{\#}$.*

Lemma 7.3 (Symbolic refinement implies behavioural refinement). *For all σ, σ', we have $\sigma \gtrsim \sigma' \implies \sigma \rightsquigarrow \sigma'$.*

Proof Sketch. For expressions this is just syntactic equality, while for value summaries this comes down to proving the correctness of the SAT solver. □

Lemma 7.4 (Completeness of symbolic execution). *For every execution of the symbolic state $H^{\#}_{\mathrm{par}}$, there exists an equivalent execution of block H_{par}. That is: for all H_{par}, we have $H^{\#}_{\mathrm{par}} \rightsquigarrow H_{\mathrm{par}}$.*

Proof Sketch. Completeness requires a bit more work, because we are given the final symbolic state and need to show that the whole block that generated it produces the same result. First we show that any instruction in the original block necessarily produces a value, which follows from the semantics of encountered expressions. From this, we can show that the execution of H_{par} in the current context must produce a state. Next, we use lemma 7.2 to show that $H^{\#}_{\mathrm{par}}$ must produce a state that is equivalent to H_{par}. Finally, because our semantics of symbolic states is deterministic, we can show that this state must be unique, therefore they must be equivalent. □

Finally, we can show the correctness of the verified scheduling implementation.

Theorem 7.5 (Scheduler Correctness). *Whenever $H^{\#}_{\mathrm{par}}$ is a symbolic refinement of $H^{\#}$, there is a forward simulation from H to H_{par}. That is: for all H, H_{par}, we have $H^{\#} \gtrsim H^{\#}_{\mathrm{par}} \implies H \rightsquigarrow H_{\mathrm{par}}$.*

Proof. This follows from lemmas 7.2, 7.3, and 7.4, and the transitivity of $\rightsquigarrow$. □

7.3 Managing Complexity in the Proof

The proof of theorem 7.5, together with the necessary additions to the Vericert back end, is as large as Vericert's original correctness proof. If one then adds the proofs of the if-conversion pass and the changes that had to be made to existing passes, Vericert with hyperblock scheduling has 16681 sloc of Coq definitions and 17426 sloc of Coq proofs according to coqwc, making it 3× larger than the original Vericert implementation. (We note, however, that our hyperblock-scheduling extension has not enlarged Vericert's trusted computing base at all.) It was therefore particularly important to take steps to manage the proof's complexity, primarily by breaking it up into reusable lemmas.

A substantial portion of the proof involves reasoning about the α function for symbolically executing instructions. As shown in fig. 8, the definition of this function is naturally broken down into smaller state-updates using the applicative interface for value summaries, <*> (from eq. (6)). Accordingly, it is desirable to formulate lemmas that follow the same structure. For instance, if we want to show some property holds for F <*> X, we would like a lemma that breaks this down into some related properties holding for F and X separately.

However, it is not possible to reason about the behaviour of value summaries like $\{\, (\mathit{true}, (+)) \,\}$ in isolation, because our current semantics of symbolic states (fig. 9) gives no meaning to functions such as (+). Our solution is to extend the semantics with a rule that can handle any value summary.

$$\textsc{PredExprIdentity}\quad \frac{(G, e) \in s \qquad P^{\mathrm{f}} \vdash G \downarrow \mathit{true}}{P^{\mathrm{f}} \vdash s \Downarrow_{\mathcal{I}} e} \tag{12}$$

This rule achieves this by making no attempt to *evaluate* the expression e that it selects from the value summary, and instead simply returns it. (In contrast, PredExpr demands that e can be evaluated to a value v.) Hence we call this the *identity semantics* for the value summary. By using identity semantics, it becomes possible to formulate lemmas that capture the behaviour of <*>, such as:

$$(P^{\mathrm{f}} \vdash F \Downarrow_{\mathcal{I}} e) \wedge (P^{\mathrm{f}} \vdash X \Downarrow_{\mathcal{I}} e') \implies P^{\mathrm{f}} \vdash (F \mathbin{<\!\!*\!\!>} X) \Downarrow_{\mathcal{I}} e(e') \tag{13}$$

We found that only once we were able to formulate lemmas like these did the proof become feasible. Without them, it involved a number of special cases that was simply unworkable.

8 EVALUATION

Our evaluation aims to answer the following research questions:

- **RQ1:** Does adding scheduling to Vericert lead to a significant improvement in the quality of the generated hardware (in terms of area and delay)?
- **RQ2:** Is hyperblock scheduling better than naïve list scheduling?
- **RQ3:** Does adding scheduling make Vericert competitive with unverified HLS tools?
- **RQ4:** Did our design decisions (e.g. section 6.3) lead to an acceptable compilation time?

Experimental setup. Following Herklotz et al. [25] and Six et al. [40], we evaluate our work using PolyBench/C [36]. For each benchmark, the resulting Verilog hardware design was simulated using Verilator to get the total cycle count. Each design was synthesised, placed, and routed onto a Xilinx series 7 FPGA (part number: `xc7z020clg484-1`) using Vivado to get its total area and its estimated maximum frequency. We then calculated total execution time = $\frac{\text{total clock cycles}}{\text{maximum frequency}}$. This is a minimum execution time, because in practice all designs will only run at 100MHz.

Figure 10 visualises the results of simulation and synthesis of the PolyBench/C benchmark. We use default Vericert (Vericert-original), Vericert with list scheduling (Vericert-list), and Vericert with hyperblock scheduling (Vericert-hyperblock). We also use the state-of-the-art open-source HLS tool Bambu [19] in two modes: one where all default optimisations are enabled (Bambu-default), and one where as many optimisations as possible are disabled (Bambu-no-opt). Several optimisations are built into Bambu though and cannot be disabled, such as list scheduling and loop flattening.

Answering RQ1. To assess whether adding scheduling to Vericert leads to better hardware designs, we compare the hardware produced by Vericert-original with that produced by Vericert-hyperblock. We see that, on average, hyperblock scheduling leads to hardware that requires only 0.48× the time to execute a benchmark (fig. 10a). This can be attributed to the scheduled hardware requiring only 0.46× the number of cycles (fig. 10b) and the minimum clock period being nearly identical between the scheduled and unscheduled designs (fig. 10c). This improvement is unsurprising given that Vericert-original only executes a single instruction per clock cycle. In terms of area (fig. 10d), hyperblock scheduling leads, on average, to a slight increase in area of 23.6%. This can mostly be attributed to the additional circuitry needed to check the value of predicates and gate instructions. However, in some cases, like floyd-warshall, the area increase is quite egregious, which can be attributed to some back-end optimisations like operation fusing not triggering because the Verilog code does not match a specific pattern the synthesis tool recognises. Tweaking the syntax of the generated code should allow us to produce designs with similar area to those of Vericert-original.

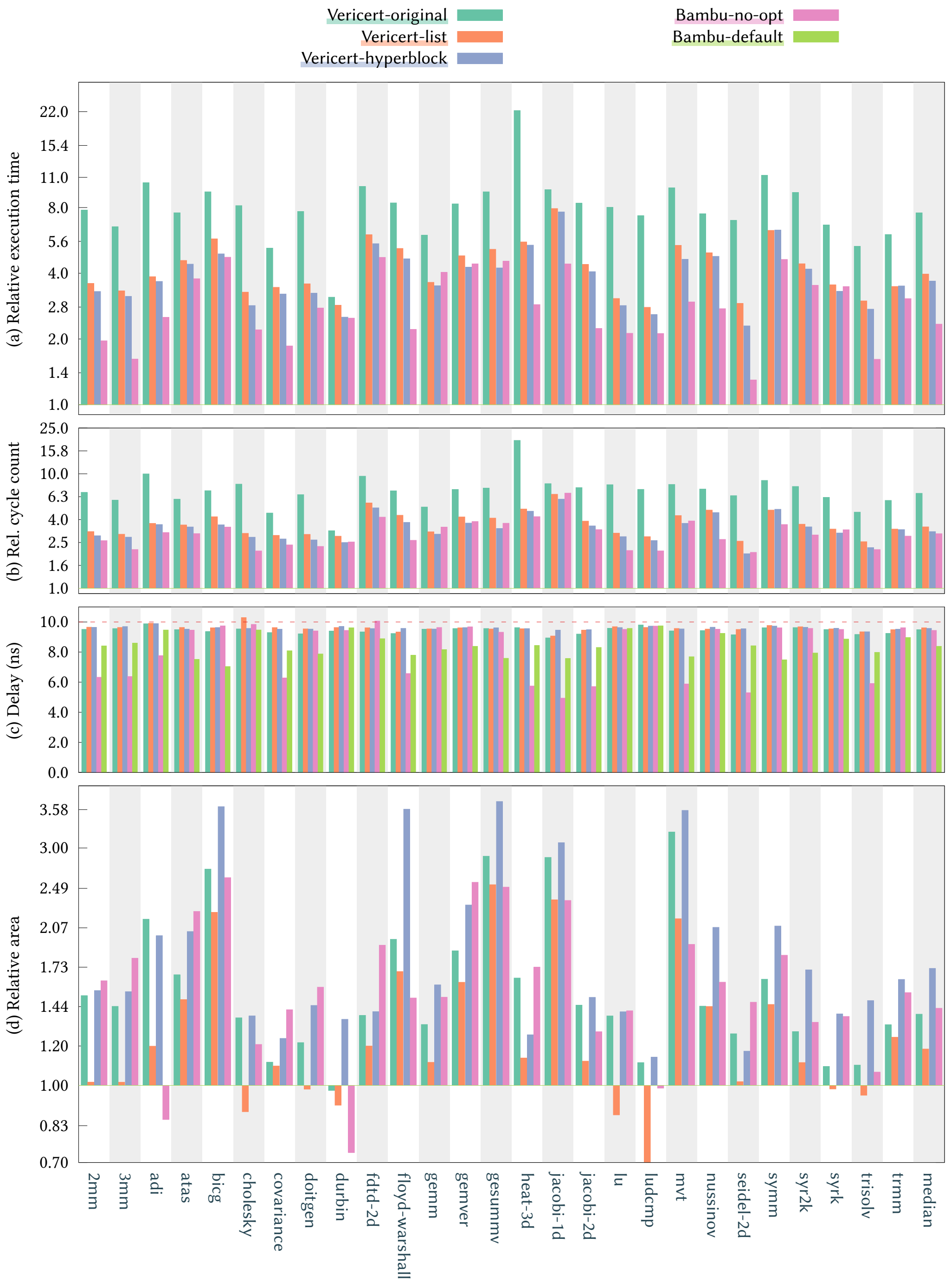


Fig. 10. Results of simulating and synthesising the PolyBench/C benchmark suite using a range of HLS tool configurations. All measurements are relative to Bambu-default except the greatest critical path delay, which is given absolutely. The dashed red line in that graph corresponds to the target clock with a period of 10ns, or a frequency of 100MHz.

Answering RQ2. Hyperblock scheduling is considerably more complicated to implement and verify than list scheduling because it requires if-conversion to combine basic blocks into hyperblocks, as well as predicate-aware scheduling. If we omit if-conversion entirely (hence avoiding predication too), we obtain list scheduling as a special case. Does hyperblock scheduling yield enough of a performance improvement over list scheduling to justify its additional complexity?

To answer this, fig. 10 measures the hardware produced by Vericert-list. On average, list scheduling leads to hardware requiring 0.52× the total time compared to Vericert-original and 1.08× the total time compared to Vericert-hyperblock. We expect hyperblock scheduling to extend its small lead over list scheduling once the if-conversion heuristics are improved. In particular, our predictions of predicated instruction latency are currently quite conservative to ensure that timing constraints are met; improving these estimates is an active research area [37, 41, 44, 45, 48]. On the other hand, the latency of instructions without predicates is much more predictable and its estimation less conservative, leading to better overall performance. We believe list scheduling is at its best, whereas hyperblock scheduling could be greatly improved with better predictions.

In terms of area, we see that Vericert-list leads to the smallest hardware designs, on average 0.69× the size of Vericert-hyperblock designs. This can be attributed to the downstream logic synthesis tool being able to save area by optimising chained operations, such as multiply–accumulate, while not having to handle the predicates that are introduced with Vericert-hyperblock.

Answering RQ3. To assess how Vericert-hyperblock fares against unverified HLS tools, we compare it against Bambu. We see that although Vericert-hyperblock is well behind Bambu-default (its designs require 3.6× the execution time), it only performs slightly worse when compared to Bambu-no-opt (1.57× the execution time). This is encouraging because the main reason for the slow-down in execution time is the higher critical path delay – Vericert-hyperblock performs comparably with Bambu-no-opt in terms of cycle count, with its designs needing only 1.04× the cycles. Cycle count is under the direct control of the scheduler, and therefore shows the effect of the scheduler most clearly, whereas the critical path delay is sensitive to which optimisations the downstream synthesis tool decides to perform. As mentioned previously, estimating the delay of operations and predicting when downstream optimisations will fire is an active research area and is currently implemented conservatively. For example, even Bambu-no-opt failed to predict the delay of the critical path correctly for the fdtd-2d benchmark, and failed timing for it, as did Vericert-list on the cholesky benchmark. In terms of area, Vericert-hyperblock designs are 1.7× larger than Bambu-default and 1.2× larger than Bambu-no-opt, however, this is tightly linked to the fact that some chaining optimisations are currently missed by the downstream synthesis tool. Tweaking the representation of predicated instructions could improve the area significantly.

Answering RQ4. To assess whether Vericert-hyperblock has acceptable compilation times, we also compare it against Bambu. Compilation times did not deviate for Bambu, all of them being around 3s mainly due to long startup costs. Vericert-hyperblock compiled each benchmark in 0.9s, also without much variation, showing that verification was not overly costly. As for whether our design decisions led to these compilation times: we remark that if we disable the 'final-state predicates' innovation that we introduced in section 6.3, none of the benchmarks compile within a few minutes and eventually the machine runs out of memory.

9 RELATED WORK

The most closely related works to ours are those by Tristan and Leroy [43] and by Six et al. [40], so we begin this section by recapping our main points of similarity and difference.

Tristan and Leroy [43] were the first to propose adding scheduling to a verified compiler, and we adopt their method for validating schedules – running symbolic execution before and after scheduling and comparing the resultant symbolic states. Their scheduler only *reorders* instructions, so syntactic equality suffices for comparing symbolic states, whereas our scheduler can also *modify* instructions (by manipulating predicates). This means that our state comparisons are more involved, and we turn to a SAT solver to help resolve them (section 6.2). Tristan and Leroy also devised the use of constraints to prevent the scheduler introducing undefined behaviour; we adopt this technique too, taking care to extend it to handle predicates in such a way that valid schedules can still be validated (section 6.4). We remark that a direct empirical comparison with Tristan and Leroy's work is not feasible because their method was implemented for an old version of CompCert and was not incorporated into its main branch.

Six et al. [40] formalise superblock scheduling, which is a form of trace scheduling that is well-suited for VLIW processors. Hyperblock scheduling is more general than superblock scheduling and is well-suited to our application domain, HLS. Six et al.'s scheduler reorders instructions, and also combines them where it is advantageous to do so, reasoning only about registers that are live at the end of the block, so comparing symbolic states is more involved than it was for Tristan and Leroy, but it still does not require a SAT solver because there are no predicates to reason about. A direct empirical comparison between our scheduler and Six et al.'s is difficult because they have different targets and Six et al.'s is based on an incompatible fork of CompCert called CompCertKVX.

Outside of the realm of mechanised proof, most HLS tools implement some form of static scheduling. For instance, AMD Vitis HLS [3], LegUp [11, p. 60], Google XLS [21, line 112] and Bambu [18, line 35] all employ SDC-based hyperblock scheduling. Other HLS tools, such as Dynamatic [31], defer scheduling decisions until runtime. It is notable that these unverified tools tend towards fewer, larger passes, where several optimisations are packed into 'scheduling'. This minimises the number of times the solver needs to be invoked, and gives it the best chance of finding a global optimum. Verified tools such as CompCert and Vericert, on the other hand, tend towards more, smaller passes that are individually feasible to prove correct.

Several HLS tools are associated with translation validators, either for an individual scheduling pass [14, 32, 46] or for the HLS tool as a whole [39, 42]. These typically work by establishing equivalence at the level of state machines, whereas we take the approach of comparing symbolic states, because Tristan and Leroy [43] showed it to be viable in the context of a verified compiler. Unlike our work, none of these translation validators provide mechanised proofs of equivalence, so might produce false positives. Google XLS [22] performs scheduling on a dataflow language, which may also be a useful intermediate representation to formalise a scheduler on directly (as opposed to validating each schedule produced). Formalisation of dataflow languages within CompCert is being actively worked on [16, 23], which may make this feasible.

10 CONCLUSION

We have presented the first verified implementations of general if-conversion and hyperblock scheduling, and incorporated them into the Vericert verified HLS tool. The practical value of this work is that it extends verified HLS from the prototype of Herklotz et al. [25] into an increasingly practical tool. On the more conceptual side, our work may prove useful to those implementing other optimisation passes in a verified compiler using solver-powered validation.

There are opportunities for further performance improvements by tweaking the if-conversion heuristics and the implementation of the scheduler. Both of these should be relatively straightforward because neither affects the correctness proof. Longer term, we hope that this implementation of hyperblock scheduling and the new hyperblock intermediate languages provide a base for future HLS-specific optimisations. In particular, the main remaining optimisations that are needed for Vericert to become truly competitive with Bambu are modulo scheduling [47] (which would enable loop pipelining) and memory partitioning.

ACKNOWLEDGMENTS

We would like to thank George Constantinides, Quentin Corradi, Alastair Donaldson, Lana Josipović, Michalis Pardalos, and the anonymous reviewers for their helpful feedback. This work was financially supported by the EPSRC via the Research Institute for Verified Trustworthy Software Systems (VeTSS) and the IRIS Programme Grant (EP/R006865/1).

ARTEFACT AVAILABILITY

A software artefact of Vericert with hyperblock scheduling, as well as the raw data used in the evaluation section, has been archived [26]. Vericert is open source and is being developed on GitHub:

https://github.com/ymherklotz/vericert

REFERENCES

[1] AbsInt. 2019. CompCert release 19.10. https://www.absint.com/releasenotes/compcert/19.10/.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[3] AMD. 2023. Vitis Forums. https://bit.ly/vitisifc (accessed 2023-06-02). Relevant quote from AMD: "If-Conversion aims to convert a sequence of blocks into a single block for better optimization result.".

[4] AMD. 2023. Vitis High-level Synthesis. https://bit.ly/41R0204 (accessed 2023-05-21).

[5] Kenneth R. Baker and Dan Trietsch. 2019. *Principles of sequencing and scheduling* (second edition. ed.). Wiley, Hoboken, NJ, USA.

[6] Thomas Ball and James R. Larus. 1993. Branch Prediction for Free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/155090.155119

[7] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 4 (mar 2014), 35 pages. https://doi.org/10.1145/2579080

[8] Michel Berkelaar. 2010. `lp_solve` v5.5. https://lpsolve.sourceforge.net/5.5/.

[9] Mihai Budiu and Seth Copen Goldstein. 2002. Compiling Application-Specific Hardware. In *Field-Programmable Logic and Applications, Reconfigurable Computing Is Going Mainstream, 12th International Conference, FPL 2002, Montpellier, France, September 2-4, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2438)*, Manfred Glesner, Peter Zipf, and Michel Renovell (Eds.). Springer, Berlin, Germany, 853–863. https://doi.org/10.1007/3-540-46117-5_88

[10] Timothy J. Callahan and John Wawrzynek. 1998. Instruction-Level Parallelism for Reconfigurable Computing. In *Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm, 8th International Workshop, FPL'98, Tallinn, Estonia, August 31 - September 3, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1482)*, Reiner W. Hartenstein and Andres Keevallik (Eds.). Springer, 248–257. https://doi.org/10.1007/BFb0055252

[11] Andrew Canis. 2015. *LegUp: open-source high-level synthesis research framework*. phdthesis. University of Toronto.

[12] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. 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)*. Association for Computing Machinery, 33–36. https://doi.org/10.1145/1950413.1950423

[13] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. 1991. Using Profile Information to Assist Classic Code Optimizations. *Journal of Software: Practice and Experience* 21, 12 (1991), 1301–1321. https://doi.org/10.1002/spe.4380211204

[14] R. Chouksey and C. Karfa. 2020. Verification of Scheduling of Conditional Behaviors in High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 7 (2020), 1638–1651. https://doi.org/10.1109/TVLSI.2020.2978242

[15] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *2006 43rd ACM/IEEE Design Automation Conference*. 433–438. https://doi.org/10.1145/1146909.1147025

[16] Delphine Demange, Yon Fernández de Retana, and David Pichardie. 2018. Semantic reasoning about the sea of nodes. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, 163–173. https://doi.org/10.1145/3178372.3179503

[17] John R Ellis. 1985. *Bulldog: A compiler for VLIW architectures*. Ph. D. Dissertation. Yale.

[18] Fabrizio Ferrandi. 2014. PandA-Bambu release notes. https://github.com/ferrandi/PandA-bambu/blob/04123a3edb37040db52c44f6591006391cf4a90b/src/frontend_analysis/IR_manipulation/predicate_statements.cpp#L35 (accessed 2023-11-16).

[19] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110

[20] Joseph A. Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.* C-30, 7 (1981), 478–490. https://doi.org/10.1109/TC.1981.1675827

[21] Google. 2023. XLS: Accelerated HW Synthesis. https://github.com/google/xls/blob/dde7095ff1050b09c37cb44d1977bff1af8de050/xls/scheduling/mutual_exclusion_pass.h#L112 (accessed 2023-11-14). The XLS scheduler refers to using an SMT solver to merge mutually exclusive nodes.

[22] Google. 2024. Google XLS. https://google.github.io/xls/ (accessed 2024-03-21).

[23] Yann Herklotz, Delphine Demange, and Sandrine Blazy. 2023. Mechanised Semantics for Gated Static Single Assignment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, MA, USA) *(CPP 2023)*. Association for Computing Machinery, 182–196. https://doi.org/10.1145/3573105.3575681

[24] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021. An Empirical Study of the Reliability of High-Level Synthesis Tools. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 219–223. https://doi.org/10.1109/FCCM51124.2021.00034

[25] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 117 (10 2021), 30 pages. https://doi.org/10.1145/3485494

[26] Yann Herklotz and John Wickerson. 2024. Artefact: Hyperblock Scheduling for Verified High- Level Synthesis. Zenodo. https://doi.org/10.5281/zenodo.10808233

[27] Gregory Littell Hopwood. 1978. *Decompilation.* Ph. D. Dissertation. University of California, Irvine. AAI7811860.

[28] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1993. *The Superblock: An Effective Technique for VLIW and Superscalar Compilation.* Springer US, 229–248. https://doi.org/10.1007/978-1-4615-3200-2_7

[29] IEEE. 2024. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. (2024), 1–1354. https://doi.org/10.1109/IEEESTD.2024.10458102

[30] Intel. 2023. High-level Synthesis Compiler. https://intel.ly/2UDiWr5 (accessed 2023-06-23).

[31] Lana Josipovic, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, Jason Helge Anderson and Kia Bazargan (Eds.). ACM, 127–136. https://doi.org/10.1145/3174243.3174264

[32] C Karfa, C Mandal, D Sarkar, S R. Pentakota, and Chris Reade. 2006. A Formal Verification Method of Scheduling in High-level Synthesis. In *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED '06)*. IEEE Computer Society, 71–78. https://doi.org/10.1109/ISQED.2006.10

[33] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* 43, 4 (2009), 363. https://doi.org/10.1007/s10817-009-9155-4

[34] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective Compiler Support for Predicated Execution Using the Hyperblock. *SIGMICRO Newsl.* 23, 1-2 (12 1992), 45–54. https://doi.org/10.1145/144965.144998

[35] Barry M. Pangrle and Daniel D. Gajski. 1987. Design Tools for Intelligent Silicon Compilation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6, 6 (1987), 1098–1112. https://doi.org/10.1109/TCAD.1987.1270350

[36] Louis-Noël Pouchet. 2020. PolyBench/C: the Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[37] Carmine Rizzi, Andrea Guerrieri, and Lana Josipović. 2023. An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247686

[38] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, 842–853. https://doi.org/10.1145/2786805.2786830

[39] Siemens. 2021. Catapult High-Level Synthesis. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/ (accessed 2023-11-14).

[40] Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse, and Nicolas Nardino. 2022. Formally Verified Superblock Scheduling. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022)*. Association for Computing Machinery, 40–54. https://doi.org/10.1145/3497775.3503679

[41] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. 2015. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. Association for Computing Machinery, 190–199. https://doi.org/10.1145/2684746.2689063

[42] Andreas Tiemeyer, Tom Melham, Daniel Kroening, and John O'Leary. 2019. CREST: Hardware Formal Verification with ANSI-C Reference Specifications. abs/1908.01324 (2019).

[43] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, 17–27. https://doi.org/10.1145/1328438.1328444

[44] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415657

[45] Hanyu Wang, Carmine Rizzi, and Lana Josipović. 2023. MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323639

[46] Youngsik Kim, S. Kopuri, and N. Mansouri. 2004. Automated formal verification of scheduling process using finite state machines with datapath (FSMD). In *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*. 110–115. https://doi.org/10.1109/ISQED.2004.1283659

[47] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 211–218. https://doi.org/10.1109/ICCAD.2013.6691121

[48] Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. 2014. Fast and effective placement and routing directed high-level synthesis for FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, 1–10. https://doi.org/10.1145/2554688.2554775

Received 2023-11-16; accepted 2024-03-31