An HLS algorithm for the direct synthesis of complex control flow graphs into finite state machines with implicit datapath

Jean-Christophe Le Lann
ENSTA Bretagne Lab-STICC UMR 6285, Brest, France
Email: jean-christophe.le_lann@ensta-bretagne.fr

***Abstract*—In this paper, we introduce an efficient algorithm for automating the direct transformation of a control flow graph (CFG) into a synthesizable finite-state machine with implicit datapath (FSMD). In our opinion, this transformation has not received sufficient attention: although the passage of a CFG to FSMD is mentioned in many textbooks on digital system design, and presented as trivial, to our knowledge it has never been explicitly formulated nor automated. Our experience shows, moreover, that this process is trickier than claimed. We believe our algorithm can become a key transformation for high-level synthesis (HLS) of control-dominated applications presenting a low level of instruction-level parallelism. Our paper presents the algorithm in detailed procedural form. Experimental measurements carried out on synthetic benchmarks shows its effectiveness.**

I. Introduction

High-level synthesis (HLS) [1] is now a mature and widespread technique for designing digital circuits. It involves transforming a source program, usually written in C or any other imperative language, into a circuit described at register transfer level (RTL). HLS thus relieves the designer of the task of defining the micro-architecture, which is considered error-prone at this level of design. What's more, HLS enables the designer to explore different solutions for far more complex algorithms than can be handled by paper and pencil. The adoption of HLS in most industrial design-houses has been achieved thanks to the intensive and continuous work of the academic community [2]. However, there has been a strong focus on a particular type of applications, namely those with a high degree of instruction parallelism (ILP). These applications enable HLS compilers to exhibit data-flow graphs (DFG), on which the algorithms are now particularly effective: HLS compilers are able to propose near-optimal solutions in terms of the number of resources used or the number of computation cycles. These synthesis capabilities are particularly impressive and irreplaceable for large DFGs.

However this focus and progresses have been to the detriment of control-intensive applications. In this second type of applications, we find a large number of conditional instructions, as well as loops (notably non-unrollable loops) carrying out waiting operations (synchronization on inputs or outputs) interwoven with conventional calculations. For such applications, HLS compilers struggle to run their traditional algorithms: for example, the classic phase of scheduling the instructions present in the DFG on different control-steps is inoperative here, by lack of deep DFGs. This type of control-intensive applications is much more widespread than initially imagined. For example, they intervene at the periphery of regular calculation kernels, in order to retrieve data according to protocols that are often complex. Among them are the protocols associated with video decoders (H264, for example), which have a real syntax requiring the development of complete and complex parsers [3], which will feed SIMD-like accelerator kernel dedicated to image reconstruction [4]. More generally, they also appear during packet processing in network-oriented applications [5]. These parsers require few calculations: only a few counters need to be updated, as and when required. In general terms, this means that there are applications that can make many complex decisions without complex calculations. This absence of complex calculations makes conventional HLS practically useless, and calls for alternative way of thinking about RTL generation.

In this article, we propose an algorithm that, given a control-intensive application, transforms its control flow graph to a finite state machine form at the RTL level *directly*, without running conventional HLS passes. Although such processes have been mentioned in the literature, to our knowledge no effective formulation in executable algorithmic form has been proposed.

Our paper is structured as follows. Section II provides some definitions and a formulation of the problem. Section III details the algorithm using several pseudo-codes. Section IV presents experiments carried out to validate the algorithm and particularly its performance. Sections V proposes different related works. Section VII finally concludes our paper.

II. Definitions and problem statement

A. Control flow graphs

Control flow graphs (CFG) serve as a classical compiler intermediate representation (IR) for representing and analyzing program behaviors, enabling various compiler optimizations, such as dead code elimination, loop optimization, and register allocation. Hereafter, the terms CFG and IR may be used interchangeably. A CFG can be processed as a directed graph $G = (V, E)$ where:

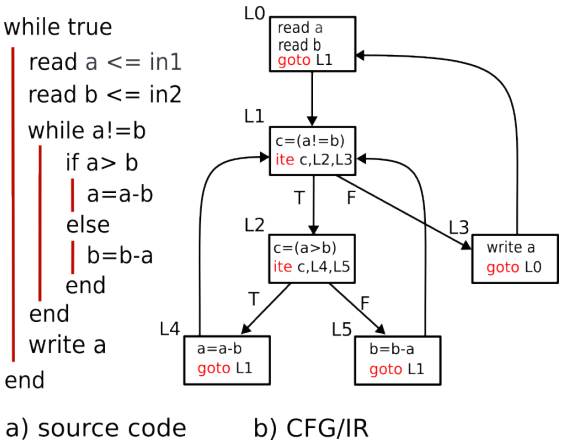


Fig. 1. GCD source code example, together with its CFG acting as compiler intermediate representation

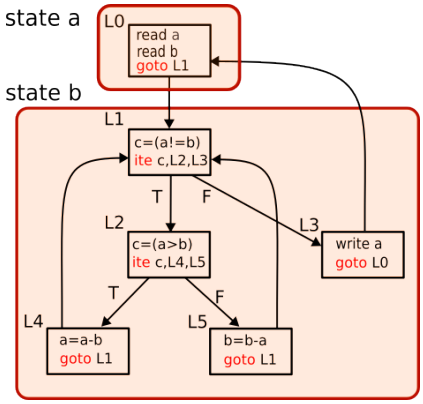


Fig. 2. Problem statement : given a CFG with n basic blocks (here $n = 6$) from figure 1, we look for the FSM covering the CFG using a minimum number of states (here 2) that respect synchronous digital design rules.

- V is a set of vertices representing basic blocks or control flow nodes in the program.
- E is a set of directed edges representing control flow transitions between basic blocks. Each edge $(v_i, v_j) \in E$ indicates that control can flow from basic block v_i to basic block v_j. The edge from v_i to v_j may be labeled as $v_i \xrightarrow{\alpha} v_j$ with a boolean condition value $\alpha \in \{\texttt{true}, \texttt{false}\}$, in the case where basic block v_i contains a conditional branching instruction .

The CFG satisfies the following properties:

1) Single entry point: there exists a unique vertex, typically denoted as the entry node, representing the starting point of the program's execution. In the sequel, CFG entry points will be drawn at the top of the figures.
2) Single exit Point: there may exist a unique vertex, typically denoted as the exit node, representing the termination point of the program's execution. In the case of reactive applications, that continuously interact with their environment, this exit point may not exist (looping behaviors).

B. Basic blocks and superblocks

A *basic block* itself is a sequence of sequential instructions with a single entry point at the beginning and a single exit point at the end. It contains no branches, except possibly at the end : in the sequel, we will resort to `goto` for an unconditional jump to next labeled basic block and `ite` (if-then-else) for a conditional jump to two labeled basic blocks, depending on a boolean value.

The literature also evokes the notion of *superblocks*. A superblock is an extended basic block that may contain multiple basic blocks. Superblocks are designed to capture linear sequences of code execution (sometimes called 'traces'). A superblock ends when the flow of control hits a conditional jump like `ite` or any control flow instruction that introduces multiple potential paths of execution, such as a switch statement or a function call. In this article, we will aim to create groupings of chained basic blocks, but we won't talk about superblocks, as this chaining can indeed contain basic blocks pointed to by such a `ite` branching instruction.

C. Overview of the HLS process

Together, the CFG and DFGs form a CDFG (control and data flow graph). In classic HLS synthesis, instruction parallelism is mainly found in the basic blocks: the compiler seeks to establish a data flow graph from these instructions. As the DFG has no cycles, it is often referred to as a DAG (direct acyclic graph). It could also be noted that this data flow graph is potentially unconnected (meaning that several DAGs can emerge from a single basic block). Based on resource constraints expressed in number of functional units (FU) allocated, DFGs nodes are then scheduled on control-steps (c-steps) and mapped to these FUs. DFG edges are similarly mapped to either registers or wires, depending on operation chaining. Datapath controller generation ensures that the order of scheduled operations is respected, and that data is routed from registers to functional units. This requires the creation of as many intermediate states as there are control steps (c-steps). But this same controller must, naturally, ensure the execution of the application's control flow: most generally, the controller ensures inter-basic block sequential chaining, by allocating at least one state per basic block. In traditional HLS, the general structure of the controller thus reflects the structure of the CFG.

D. Problem statement

We will now look at the degenerate case where the DFGs are particularly shallow and show little parallelism, and where, conversely, the CFG is predominant. The previous HLS process is then visibly naive: such a naive synthesis leads to the generation of a controller that now strictly reflects the initial CFG structure, which is of no interest when looking for hardware acceleration. To help us understand this, let's look at the CDFG of the very trivial Euclide GCD (greatest common divisor) calculation example in Figure 1. None of the contents of the basic blocks are parallel, and our previously described classical HLS procedure inevitably leads to a controller with 6 states (one per basic block). However, we are well aware that it is possible to describe a much clever automaton as shown in the figure 2, using the fact that it is perfectly possible to perform conditional operations in combinatorial logic, leading

to Mealy-style FSM. This automaton now has only 2 states. We can now formulate the problem we're seeking to address as follows: **given the CFG of an application, how many states are required for its execution as a synthesizable finite state machine ?**

E. Implicit FSMD model as process target

The previous GCD example led us to modify the result of the behavioral synthesis: we authorized the generation of an FSM whose states operate *directly* on the variables of our application, rather than controling an external datapath. In reality, this design model is more natural for RTL designers, who do not systematically separate controler and datapath whenever possible (which is still the case when no resource sharing is involved). This universal model is an extended state machine, which performs calculations *in situ*. As such, this style of FSMD is sometimes refered as *implicit* as the datapath ('D' of FSMD) is not explicitly isolated [6]. A formal description of this model can be found in [7]. In concrete terms, if we refer to VHDL or Verilog code, this means that the automaton described not only has state transition logic, but also combinatorial conditional assignments that modify signals and registers. Our key observation is that these conditionals can be deeply nested and as complex as desired, at least as long as they don't burden the clock frequency. Both assignments and state changes can be described within these nested conditionals. These conditionals, which would necessarily involve a change of state in a classical HLS, can here be absorbed by combinatorial logic, within a smaller number of states. Figure 3 gives an example of the textual FSM generated by our compiler (named Archipel) for the GCD example. Note also that despite their intuitive nature, FSMDs

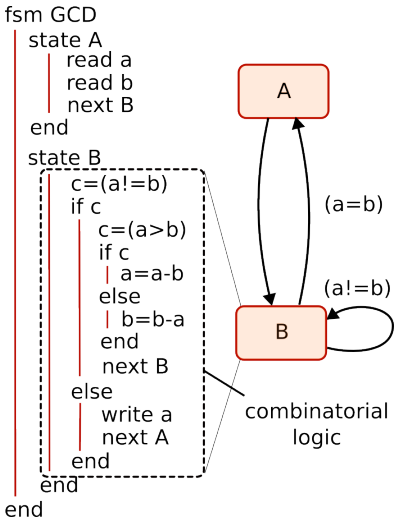


Fig. 3. Textual FSM generated for GCD example from figure 1, as output of our compiler. Syntax resembles VHDL or Verilog and makes it easy to generate both (VHDL in our case). Nested `if-else` statements appearing here have been easily regenerated from `ite` and `goto` basic block statements within CFG/IR.

do not dispense with the usual precautions when it comes to digital design: in particular, combinatorial loops are obviously

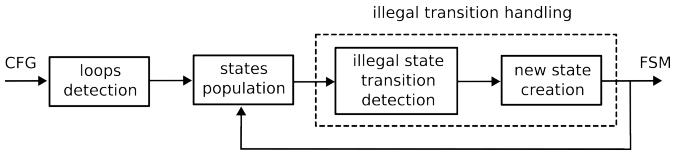


Fig. 4. Algorithm flow.

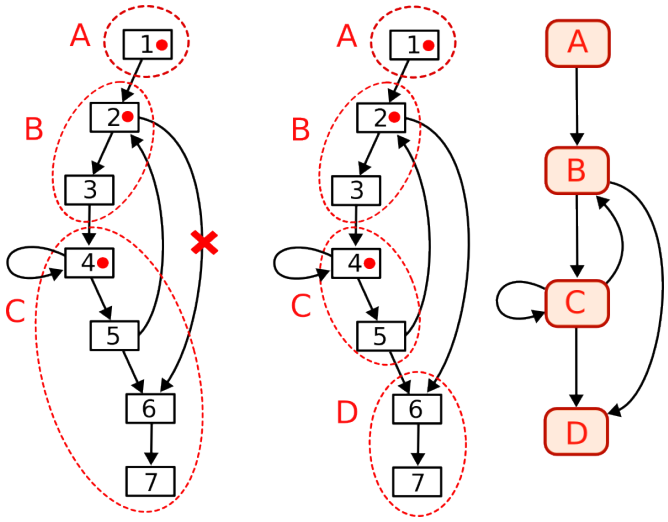


Fig. 5. Algorithm illustration. On the left : loop detection (red dots), state population (dotted ovals) and illegal transition detection (red cross). In the center : supplemental state D created to fix illegal transition. On the right : final FSM structure.

illegal. This remark is the very starting point of our algorithm, proposed in next section.

III. Algorithm Proposed

A. Overview

Our top-level algorithm is presented in pseudo-code Algorithm 1 and illustrated in Figure 4. It is divided into three main steps : loop detection, states population and illegal transition handling. Its execution can be followed again as depicted in Figure 5 on an illustrative input CFG.

Loop detection step Our algorithm first seeks to detect the basic blocks to which execution paths loop. If ignored, these structural loops between basic blocks would also generate loops in the combinatorial logic of the FSM, which is of course prescribed. Each of these basic blocks then represents the first basic block of a distinct state of the FSM under construction. We will call such a basic block a "*starter*" of the state. The visit of the CFG itself follows a depth-first seach (DFS) algorithm [8]. Figure 5 shows an illustrative case. The red dots indicate the determination of these starters (1,2 and 4 basic blocks). As the CFG entry, basic block labelled 1 naturally starts a state.

State population step From every starter, associated with a state, the CFG is traversed again in a depth-first manner, until another starter is hit. When hitting such a starter, it means that we have reached the limit of the current state and the switching to a new state. Until reaching this limit, the basic blocks visited

are mapped to the current state. These first groupings of basic blocks are shown dotted on the figure. In our algorithm, this simple grouping process is called the *population phase*. In our example, we finally obtain 3 distinct states, A, B and C, from 7 basic blocks.

Illegal transitions handling step Our algorithm continues with an mandatory third step, which is the heart of our procedure. To understand why previous steps are not sufficient, let's take a look at the transition from 2 to 6 (marked with a cross). The associated states are B and C. Transitioning from 2 to 6 likewise is actually illegal : in terms of state transition, it would imply that we can transit from one state to another, without passing through the basic block starter of the destination state. This RTL behavior is impossible to write in HDL code : it is not possible to transition to the "middle of a state". To fix this situation, an additional state (named D in the example) is added, whose starter is the destination basic block. In our example, after a population phase (similar to step 2) , the new fourth state D now includes basic blocks 6 and 7.

Iteration step As the creation of this last state may itself invalidate transitions processed previously, previous step must be iterated. Our example of figure 5 does not necessitate any such iterations, but our next experiments on synthetic benchmarks will make them mandatory. We notice that this iteration step has an acceptable complexity cost: indeed, the initial grouping of basic blocks, during the first two phases of the algorithm, tends to facilitate the quick detection of these illicit transitions. This is verified experimentally in section IV.

This algorithm makes it possible to establish the final FSM, represented fig 5 for our illustrating example : we finally get 4 states, from a CFG consisting of 7 basic blocks.

B. Pseudo-code

To maximize its reuse, our algorithm pseudo code is presented below in its entirety.

Algorithm 1 Top-level CFG to FSMD

Input: a CFG
Output: a FSM

```
1: call loop detection procedure                ▷ Algorithm 2 & 3
2: set retry to true
3: while retry do
4:     call populate states procedure           ▷ Algorithm 4 & 5
5:     call detect_illegal_transitions          ▷ Algorithm 6 & 7
6:     if number of states differs then
7:         set retry to true
8:     else
9:         set retry to false
10:    end if
11: end while
12: return FSM
```

IV. Experiments

A. Evaluation of algorithm complexity

We conducted a number of experiments to measure the efficiency of our algorithm, and to deduce an asymptotic

Algorithm 2 Loop detection

Inputs: CFG
Output: FSM

```
1: mark all nodes as not visited
2: pick the CFG basic block entry L_0
3: create a first current state, that covers L_0
4: add current state to FSM
5: map L_0 to current state
6: call detect_loop_rec(CFG,L_0,empty path)
7: return FSM
```

Algorithm 3 Recursive **detect_loop_rec** procedure

Inputs: CFG, bb node, path
Output: updated FSM

```
1: if node already visited then
2:     if node already in current path then
3:         if node not already a state starter then
4:             add new state with node as starter
5:             map node to state
6:         end if
7:     end if
8:     return
9: else
10:    add node to visited set
11:    for all node successors do
12:        new_path=clone path
13:        add node to new_path
14:        call detect_loop_rec(g,succ,new_path)
15:    end for
16: end if
```

Algorithm 4 State population procedure

Inputs: CFG, FSM
Output: updated FSM

```
1: mark all nodes as not visited
2: for all state of the FSM do
3:     get state starter L
4:     call populate_rec(cfg,L,state)
5: end for
```

Algorithm 5 Recursive **populate_rec** procedure

Inputs: CFG, bb node, state
Output: updated FSM

```
1: return if node already visited
2: if visited node is already a state starter then
3:     return
4: else
5:     if visited node is not current state starter then
6:         map node to state
7:     end if
8:     mark node as visited
9:     for all successor succ of node in CFG do
10:        call populate_rec(cfg,succ,state)
11:    end for
12: end if
```

Algorithm 6 Iterative **detect_illegal_transitions**

Inputs: CFG, FSM
Output: updated FSM

```
1: pick CFG entry L0
2: call detect_illegal_transitions_rec(cfg,L0)
```

Algorithm 7 **detect_illegal_transition_rec**

Inputs: CFG,bb node
Output: updated FSM

```
1: get current state that maps to node
2: if node already visited then return
3: else
4:     mark node as visited
5:     for all successor succ of node do
6:         get next state that maps to succ
7:         if curr_state != next_state then
8:             get starter of next state
9:             if succ!=starter then
10:                if no state has already this starter then
11:                    create new state, starting by succ
12:                    map succ to this new state
13:                    abort illegal transitions detection
14:                end if
15:            end if
16:        end if
17:        call detect_illegal_transition_rec(cfg,succ)
18:    end for
19: end if
```

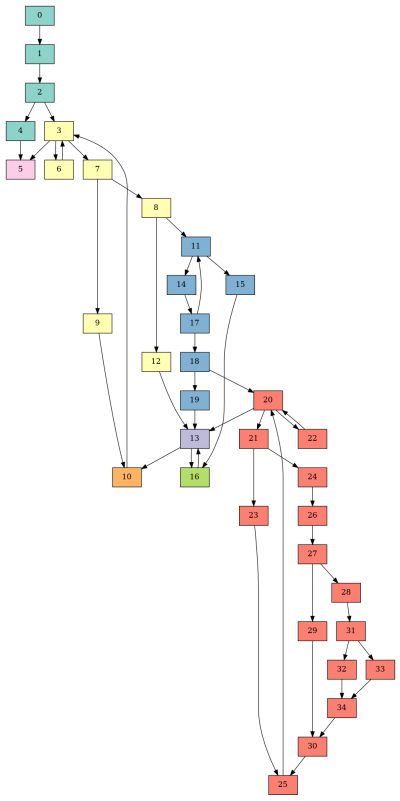


Fig. 6. Example test CFG generated, and processed by our algorithm. Same colored basic blocks indicate they belong to the same FSM state. Here eight states were found.

complexity, which is difficult to find analytically. To this end, we randomly generated a large number of CFGs and report the number of states found for each of the CFG.

These CFGs can contain a variable number of basic blocks, transitions (`ite` and `goto`) and loops. In order to provide an objective measure of the complexity of generated CFGs, we provide the cyclomatic complexity measure, due to McCabe [9]. Cyclomatic complexity quantifies the intricacy of a program's control flow graph, correlating with the difficulty of comprehending and analyzing its logic; higher complexity implies greater cognitive load for understanding code. In concrete terms, this is the simple formula $C = E - N + 2P$, where N is the number of basic blocks, E the number of arcs and P the number of graph connected components. In our case, P is always 1, because our CFGs take the form of a perfectly connected graph. The measures are reported in Figure 7.a. The measured worst case complexity is polynomial in $O(n^{1.58})$.

B. Efficiency of states gathering

Figure 7.b shows the measurement of the number of detected states as a function of the number of basic blocks present in the input CFG. The relationship is clearly linear: the directing coefficient of the linear interpolation is 3.4, which means that on average each state is capable of covering 3.4 basic blocks.

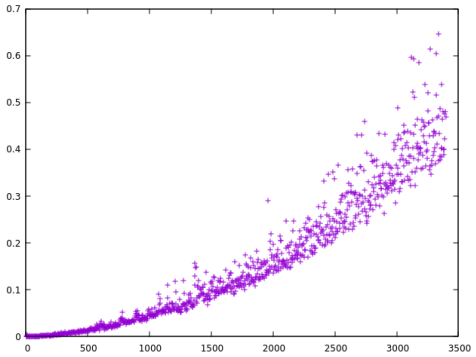
V. Related works

A. ASMD related work

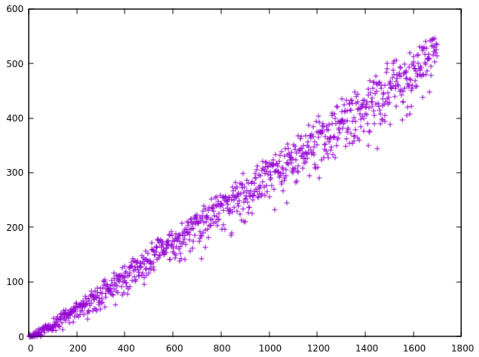
As mentioned in the abstract, many introductory books on digital circuit design allude to a process similar to the one described in our article. However, the object on which this procedure is applied is not explicitly a compiler intermediate representation like a control-flow graph : in particular, these books start from a relatively old graphical formalism called ASM or ASMD (algorithmic state machine description), originally developed by Thomas E. Osborne and made popular by Christopher R. Clare [10] in the 60-70th. For instance, this formalism is at the heart of Pr Chu's more recent series of books (for instance [6]), but can also be found in many others [11] etc. ASMD can be understood as a *flowchart* taken as entry graphical language. None of these books actually proposes an explicit algorithm for switching from ASMD or flowchart to FSMD. Barkarov's book [12] also considers such an ASMD formalism as the means to capture circuit specifications at a higher level than finite state automata (like [13]), and details how to encode them at the logic level. However, the production of the ASMD is considered to be the responsibility of the designer, but is not part of a complete HLS approach.

B. Hydrid representation of control and dataflows for HLS

Several authors have attempted to propose intermediate representations that reconcile control-flow and data-flow concerns



(a) Algorithm performance measured as time (y axis, in seconds) function of McCabe complexity (x axis). Lower is better. McCabe complexity accounts here for a growing number of basic blocks and branching behaviors. The measured worst case complexity (upper points) is polynomial in $O(n^{1.58})$



(b) Capacity of our algorithm to gather basic blocks into states : number of states detected for 1000 CFGs with a growing number of basic blocks (x axis)

Fig. 7. In each case, 1000 CFGs have been generated and processed using our algorithm prototyped as a Ruby program (a notably slow programming language).

for HLS. These alternative representations to conventional CDFGs are called "hybrid" CDFGs [1]. For example, [14] and [15] have proposed a type of fully flattened dataflow graph, whose arcs are annotated with complex boolean guards, restoring instantaneous control paths from the application's point of view. This approach, based on the complete flattening of data flows and the construction of such new hybrid control-data flow graph, also means a complete dismantling of the CFG, which is no longer manipulated as such during HLS. Dropping the CFG structure seems highly damaging, and in particular makes the generated code particularly obscure.

C. Dataflow analysis

In the field of compilation, there is of course a wealth of work on visiting the CFG for different purposes. This is particularly true of dataflow analysis. Dataflow analysis is a technique that examines how values flow through a program, essential for optimizations and dependency detection. It is divided into forward and backward analysis, determining the propagation of data from beginning to end and from points of use to origins, essential for eliminating dead code, propagating constants and detecting loops. The places where values are produced and consumed can be different basic blocks, making dataflow analysis highly dependent on analysis of the CFG structure. The work of Cytron et al. [16] formalized the SSA (Static Single Assignment) form of CFGs. In this approach, the CFG is revisited in order to rename variables so that they are assigned only once, thus facilitating downstream code analysis and optimization. This renaming can occur in basic block themselves, but control paths need to be taken into account. In particular, join points (basic block with more than one predecessor) have a strong influence on the effectiveness of this renaming and call for a judicious insertion of nodes called "phi", which allow multiplexing of the variables previously renamed. Several methods have been proposed in the literature [17], but the most effective and widely accepted is based on the determination of "dominance frontiers". As a reminder, a basic block B_1 dominates a block B_2 if and only if all paths to B_2 pass through B_1. A dominance frontier is the collection of nodes that are precisely "one edge away" from being dominated by a specific node. To put it differently, node A's dominance frontier includes node B if and only if A doesn't strictly dominate B, but A does dominate some predecessor of B. This definition maps well to our idea of changing from one state to another : our understanding is that our state starter are precisely the nodes belonging to this dominance frontier, an that the set of nodes dominated by a node n belong to the same state, starting with n. This calls for confirmation of this intuition, but also may open new ways to jointly think about control (our paper) and dataflows (works on SSA) from both a VLSI and compilers point of view : for instance, gathering basic blocks for FSMD determination like we proposed (with VLSI purpose in mind) may constitute an interesting new intermediate step while compiling plain software applications.

D. If-conversion compiler pass

We can also compare our procedure with alternative approaches, in particular the compilation procedure known as "if-conversion". The aim of the "if-conversion" process is to better deal with the presence of conditionals in programs, by transforming them into predicated instructions, ready for standard dataflow analysis[18]. This optimization enables the number of basic blocks interconnected by `ite` instructions to be reduced, by merging them: they then individually present more potential for linear execution. It is particularly useful to augment the level of instruction parallelism and simplify the workloard of branch predictors. Our FSM elaboration process automatically manipulates the conditions associated with the assignments encountered in the basic blocks, and renders them nested in the combinatorial logic. This manipulation is effectively similar to the symbolic manipulation performed by predication-based compilers. It's conceivable that these compilers could benefit from our process. However,

the comparison ends there, as the objectives for developing our FSM differ: among the interesting values of our process is the idea of returning to the RTL designer an FSM very close to the initial structure of his behavioral code. In particular, our process doesn't flatten predicates, but restores their effects by preserving their hierarchy, in a source-to-source manner.

E. Synchronous language similarities

We also noted some other interesting similarities between our approach and those studied in the context of synchronous languages [19] compilation. These languages have a formal semantics inspired by synchronous digital circuits, but aim to generate embedded software code. The compilation of Esterel in particular has generated a great deal of work, including the particularly well-documented work of Edwards at Columbia University [20] and the joint work of Berry and Potop-Butucaru at INRIA [21]. Because of its concurrent nature, Esterel requires threading, which we don't have to deal with here. However, each thread has been studied with the aim of generating fast software code for simulation purposes. The proposed technique consists in setting up a particular CFG, called GRC, and dividing it into "clusters". Once execution has started, the cluster must run without interruption. At the end of its execution, it must be possible to connect to a new cluster. This is very similar to our basic blocks grouping by states. It is worth noting that the authors found that generating software code in the form of automata was quicker than a total flattening in the form of boolean circuits, which prompts us to consider future work on simulation acceleration, which could benefit from our algorithm. We also note that the authors of [20] point out that their heuristics are not perfectly satisfactory. Here too, it will be interesting to see whether our algorithm provides an effective solution to this shortcoming.

VI. Summary and future work

In this article, we set out to clarify the direct transformation of a control flow graph (CFG) into an extended finite state machine (FSMD). The proposed technique is of interest for applications where complex decision-making is predominant over data flows. We stress that, to our knowledge, no rigorous formulation of this algorithm has ever been provided. Even if such a formulation existed in the past (a possibility we acknowledge), this formulation deserves particular renewed attention given the widespread adoption of HLS.

We mention that our prototype HLS compiler, named "Archipel", uses these FSMs as an supplemental intermediate representation that can be directly synthesized at RTL level. In the general case of compute-intensive kernels, these RTL codes result in frequencies that are obviously too low, and our compiler then triggers a classical HLS synthesis. Our current work is to study the benefits of these FSMDs as intermediate representations in conventional passes. Resource sharing should logically benefit from this pre-analysis in the form of FSMs.

Beyond pure hardware synthesis concerns, we believe that the transformation of sequential codes to finite state automata, as outlined in our paper, is of more general interest. In particular, these automata are strangely absent from most traditional compilers, or only very implicit. Our paper may give the idea of reintroducing these automata in a simple and natural way into compiler flows.

References

[1] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level synthesis: introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[3] M. Wu, Y. Chen, and C. Tsai, "Hardware-assisted syntax decoding model for software AVC/H.264 decoders," in *International Symposium on Circuits and Systems (ISCAS 2009), 24-17 May 2009, Taipei, Taiwan*. IEEE, 2009, pp. 1233–1236.

[4] T.-C. Chen, C.-J. Lian, and L.-G. Chen, "Hardware architecture design of an h.264/avc video codec," in *Asia and South Pacific Conference on Design Automation*, 2006.

[5] V. Puš, L. Kekely, and J. Kořenek, "Design methodology of configurable high performance packet parser for FPGA," in *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*. IEEE, 2014, pp. 189–194.

[6] P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. Wiley, 2008.

[7] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, "An equivalence-checking method for scheduling verification in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 556–569, 2008.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[9] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[10] C. R. Clare, *Designing logic systems using state machines*. McGraw-Hill, 1973.

[11] M. Zwolinski, *Digital System Design with VHDL (2nd Edition)*. USA: Prentice-Hall, Inc., 2003.

[12] A. Barkalov, L. Titarenko, K. Mielcarek, M. Mazurkiewicz, and E. Kawecka, *Logic Synthesis for VLSI-Based Combined Finite State Machines - Synthesis Targeting ASICs, CPLDs and FPGAs*, ser. Lecture Notes in Electrical Engineering. Springer, 2022, vol. 922.

[13] S. de Pablo, F. Martínez, L. C. Herrero, J. A. Cebrián, and S. Cáceres, "Recent advances in asm++ methodology for FPGA design," in *Proceedings of the 7th FPGAworld Conference*. New York, NY, USA: Association for Computing Machinery, 2010, p. 49–59.

[14] H.-P. Juan, V. Chaiyakul, and D. Gajski, "Condition graphs for high-quality behavioral synthesis," in *IEEE/ACM International Conference on Computer-Aided Design*, 1994, pp. 170–174.

[15] A. A. Kountouris, C. Wolinski, and J.-C. Le Lann, "High-level synthesis using hierarchical conditional dependency graphs in the Codesis system," *J. Syst. Archit.*, vol. 47, no. 3–4, p. 293–313, apr 2001.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991.

[17] F. Rastello and F. Bouchez-Tichadou, Eds., *SSA-based Compiler Design*. Springer, 2022.

[18] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983, pp. 177–189.

[19] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[20] S. A. Edwards, V. Kapadia, and M. Halasz, "Compiling esterel into static discrete-event code," in *Proc. International Workshop on Synchronous Languages, Applications, and Programs, SLAP 2004*, vol. 153, no. 4. Elsevier, 2004, pp. 117–131.

[21] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*. Springer, 2007.