# A Design Flow Engine for the Support of Customized Dynamic High Level Synthesis Flows

Marco Lattuada, Fabrizio Ferrandi

# A Design Flow Engine for the Support of Customized Dynamic High Level Synthesis Flows

MARCO LATTUADA and FABRIZIO FERRANDI, Politecnico di Milano, Italy

High Level Synthesis is a set of methodologies aimed at generating hardware descriptions starting from specifications written in high-level languages. While these methodologies share different elements with traditional compilation flows, there are characteristics of the addressed problem which require ad hoc management. In particular, differently from most of the traditional compilation flows, the complexity and the execution time of the High Level Synthesis techniques are much less relevant than the quality of the produced results. For this reason, fixed-point analyses, as well as successive refinement optimizations, can be accepted, provided that they can improve the quality of the generated designs.

This paper presents a design flow engine for the description and the execution of complex and customized synthesis flows. It supports dynamic addition of passes and dependencies, cyclic dependencies, and selective pass invalidation. Experimental results show the benefits of such type of design flows with respect to static linear design flows when applied to High Level Synthesis.

CCS Concepts: • **Hardware** → **High-level and register-transfer level synthesis**; • **Software and its engineering** → Compilers.

Additional Key Words and Phrases: High Level Synthesis, Compilation Steps

**ACM Reference Format:**
Marco Lattuada and Fabrizio Ferrandi. 2019. A Design Flow Engine for the Support of Customized Dynamic High Level Synthesis Flows. *ACM Trans. Reconfig. Technol. Syst.* 1, 1 (August 2019), 24 pages.

## 1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are configurable hardware devices which can be a good solution to solve computationally demanding problems because of their high degree of parallelism and their power efficiency. Design by hand implementations for this type of devices, however, can be an issue since they require knowledge of Hardware Description Languages (HDL), which is rare expertise. For this reason, High Level Synthesis techniques [4] have been developed: they consist of (semi)-automatic design flows that starting from high-level descriptions (e.g., C/C++ source code implementations) produce the corresponding hardware implementations. High Level Synthesis shares some characteristics with compilation flows targeting general-purpose processors: they all take as input high-level descriptions, perform a set of analyses and optimizations and produce a lower-level implementation. On the contrary, one of the main differences is the significance of the design time with respect to the quality of the produced results. The overall compilation time in traditional compilation flows is something which has great importance. For example one of the suggested rules to GNU GCC developers [21] is *"Do not add algorithms with quadratic or worse behavior, ever"*. For the same reason, the compilation flows are usually composed of fixed and static

Authors' address: Marco Lattuada, marco.lattuada@polimi.it; Fabrizio Ferrandi, fabrizio.ferrandi@polimi.it, Politecnico di Milano, Piazza Leonardo da Vinci, 32, Milano, 20133, Italy.

1936-7406/2019/8-ART
https://doi.org/

sequences of passes, so that the complexity of the compilation flow is controlled. On the opposite, the execution time required for High Level Synthesis is not considered a critical aspect because it is only a part of the design flow for the automatic generation of FPGA accelerators. Indeed, the outcome of the High Level Synthesis is the input of the rest of the Synthesis process [12] which can be up to several orders of magnitudes slower than the High Level Synthesis. For this reason, a controlled and limited increment in the complexity of the algorithms exploited in the High Level Synthesis design flow is usually accepted provided that it improves the generated solutions. The quality of the results has a larger impact in case of FPGA devices since it impacts not only on the performance of the produced solutions but also on their economic cost: a non-optimized solution could not fit in the target device requiring to be implemented on a larger and more expensive one. Finally, while a software application is typically composed of thousands of instructions and it is compiled many times during its development life-cycle, the specifications on which High Level Synthesis is applied are composed of few hundreds of instructions and are synthesized a limited number of times during their development. Relaxing the constraint on the design time does not imply only to accept more complex analyses and optimizations, but also to accept more complex design flows.

Designing, describing, and running complex customized High Level Synthesis flows is not a trivial task. State-of-the-art compilers and High Level Synthesis tools are usually based on static sequences of passes to be executed, possibly declined into different flavours. The sequences can indeed be completely fixed (e.g., [22], [15]), precomputed on the basis of the results of pre-analysis of the code (e.g., [9]), or selected after multiple evaluations of different static flows (e.g., [14]). These approaches have significant limitations since several scenarios cannot be addressed:

- Update of the set of functions to be processed.
- Fixed point analyses and optimizations composed of sequences of several passes.
- Re-execution of the design flow to exploit the information collected in the later stages.

To overcome this limit and to address these scenarios, this paper presents an approach for the design, the description, and the execution of High Level Synthesis flows where, differently from the previous approaches, the design flow is not pre-computed but it is dynamically built and updated during its execution. The proposed approach is composed of two main elements: a dynamic graph describing the design flow and an algorithm to update this graph and to select which is the next pass to be executed. The graph can be dynamically updated by:

- Adding new passes.
- Adding or removing dependencies between passes.
- Invalidating already executed passes, requiring their re-execution.

It is worth noting that identifying the best dynamic design flow for a specification (e.g., by exploiting machine learning) is out of the scope of this paper.

The rest of this paper is organized as follows. Section 2 presents the related work about design flows in compilers and in High Level Synthesis. Section 3 describes the proposed approach while a case study is presented in Section 4. Section 5 presents the experimental evaluation and, finally, Section 6 draws the conclusions of this work and presents the possible extensions.

## 2 RELATED WORK

Since the execution of complex compilation flows is not a requirement of compilers, they usually implement fixed sequential flows. For example, in GNU GCC [22] there are two main classes of passes, working on the two main intermediate representations (GIMPLE and RTL) of the compiler. The sequence of passes to be executed is statically embedded in the GNU GCC, allowing the user only to enable or disable the execution of single passes through command-line options. The execution of the same pass multiple times is made possible by explicitly adding it multiple times in the list of passes to be executed. GNU GCC does not make an explicit distinction among analysis passes and optimization passes, nevertheless it defines a set of properties associated with its intermediate representations (e.g., SSA form, loops computed, etc.). Each pass can declare which are the properties that it requires to be already computed, which are the ones it will compute and which are the ones that it will invalidate. Nevertheless, this information is not used to compute the sequence of passes to be executed, but only to verify if the input intermediate representation of each pass is consistent. The GNU GCC Plugin API [10] introduces the support to dynamic load of passes by allowing the developers to add custom passes in the compilation flow. Even in this case, the order of execution of all the passes must be known when the execution starts: the developer must statically specify where the new passes have to be inserted (before, after or in place of existing passes).

Differently from GNU GCC, LLVM [15] explicitly differentiates optimization passes, which modify the intermediate representation, and analysis passes, which do not modify the intermediate representation but which can be invalidated by optimization passes. The LLVM infrastructure proposes different classes of passes (e.g., `ModulePass`, `FunctionPass`) to work at different granularity on the intermediate representation. It is possible to specify in each pass which are the analysis passes that have to be executed before and which are the analyses that are invalidated. The LLVM pass manager collects the information about all the registered passes and then computes a static global schedule which satisfies all the constraints and minimizes the re-execution of analysis passes. The passes can be organized hierarchically: a single pass can contain a pass manager which executes a sequence of passes. Even if the computation of the sequence of passes to be executed is partially delegated to the compiler, it is still computed before the starting of the compilation flow.

Considering only a fixed order of execution of compilation passes can limit the quality of the produced results, but precompute which is the best sequence for a particular application is not a trivial task, so different approaches have been proposed to solve this issue. One approach consists of executing different complete compilation flows and evaluating the results of the different runs. However, this type of approach can significantly increase the compilation time so that different techniques have been proposed to optimize its execution time. For example, in [14] a genetic algorithm is exploited to reduce the number of evaluations that have to be performed to identify the best sequence of optimizations. On the contrary, [7] focuses on reducing the execution time required by every single evaluation by means of performance estimations obtained through virtual execution. The application is profiled once to count the number of executions of each basic block, then the delay of each basic block is estimated for each considered compilation flow.

Customization of the design flow on the basis of the analyzed specification is something partially proposed in techniques aimed at identifying the best sequence of optimizations to be applied to an application. For example, in the Milepost GNU GCC project [9], a model which correlates static characteristics of an application and best optimizations sequence is built by applying machine learning to a training set composed of benchmarks built with standard compilation flows. This model is then used to identify the sequences of optimizations to be applied to new applications, without requiring to compile them multiple times. To achieve this objective, the pass manager of GNU GCC is replaced with a simple module which queries external plugins to decide the next pass to be executed making the sequence of executed passes customizable. Even if the developed compiler infrastructure potentially allows the designer to arbitrarily change the order of execution of the different optimizations, the authors focus only on enabling and disabling optimizations. The order embedded in the compiler indeed is preserved since the detection of legal orders of optimizations would require too deep knowledge of the single passes. In the design flow engine proposed in this paper instead all the legal orders of execution of passes can be very easily computed since each pass has to declare which are its prerequisites. [1] proposed a framework for the selection of the optimizations sequence. This framework extends the previous approaches by considering dynamic target-independent features for characterizing the applications and by considering Bayesian networks to select the different optimization sequences to be considered. However, even in this framework differently from the proposed approach, multiple compilation flows have to be evaluated and the single compilation flow cannot be modified during its execution.

The number of different permutations of optimizations is exponential, but only a limited subset of them is legal and is profitable at least for some applications. To describe this subset, [19] uses a graph whose nodes correspond to compilation passes and whose edges represent the probability of executing the two connected passes in sequence. All the paths in the graph, even the cyclic ones which are shorter than a fixed threshold, represent the possible compilation flows to be considered. Each of these paths is characterized by a probability obtained by composing the probability of the single edges. A limited number of paths is randomly selected and the corresponding compilation flows are executed evaluating the obtained results. The main advantage of this approach with respect to the other methodologies for the selection of the best compilation flow is that it significantly reduces the design space to be considered by limiting the search to the admissible and potentially profitable sequences, allowing design space exploration to find better solutions in less time. Despite the use of a graph, the approach presented by Nobre et al. is significantly different from the approach described in this paper. The graph adopted in the proposed solution indeed represents the single current design flow and not a collection of them since only a single synthesis flow is executed to generate the final solution. Moreover, the proposed graph is not static: during the single design flow execution, it is iteratively updated on the basis of the outcomes of the analyses and the optimizations applied to the currently synthesized specification.

Most of the existing High Level Synthesis design flows rely on existing compilers and their static management of passes sequences to implement the synthesis flow. For example, ROCCC [23] exploits the compiling infrastructures of SUIF [24] and LLVM to perform the High Level Synthesis. The LLVM pass manager is also integrated into other High Level Synthesis tools like Vivado HLS [25], xPilot [3], and Leg-Up [2]. In the last, the analysis and optimization passes targeting FPGA devices and a new back-end generating HDL description have been added to the standard LLVM compilation flow, without modifying its pass manager. The LLVM pass manager instead has been slightly modified in [5] to execute a static arbitrary sequence of passes and in [13] to iteratively built custom sequences of passes. The authors propose to generate a set of different sequences of LLVM passes to optimize the intermediate representation which is used as input of the actual High Level Synthesis flow. Different methods are used to generate the sequences: in [5] random search, genetic algorithm, and three heuristics are considered, while in [13] three other heuristics are presented. These heuristics build the sequences of passes of the design flow by iteratively choosing a pass and adding it to the sequence if the new sequence produces better results than the current best one. The main limitation of this approach is the execution time required to generate and to evaluate all the solutions. The authors of [13] state that even by considering the fastest heuristic (i.e., the one which generates the smallest number of sequences to be evaluated) and even by considering the number of clock cycles as the objective to be optimized (which can be fast retrieved by exploiting a fast simulator working on preliminary results of High Level Synthesis), the execution time of the whole High Level Synthesis is 125 times larger than the default flow. Indeed, at each iteration of the algorithm, a new sequence is computed, it is applied to the initial intermediate representation, and its effects are evaluated by means of fast simulation. Moreover, the described design space exploration cannot exit from local minima since it does not accept worse solutions. The optimization of the frequency or the area would instead require applying logic synthesis and place-and-route to evaluate the generated solutions, incrementing by order of magnitudes the execution time of the whole design flow. On the contrary, in the approach proposed in this paper, a single customized design flow is applied to each specification, with a very limited execution time increment. Obtaining the same type of results exploiting only the current version of the pass managers of GCC and LLVM without any further modification is not possible. Nevertheless, the presented technique can still be integrated into such type of infrastructure by modifying how the passes are managed. Details of this possibility will be provided in Section 3.3.

## 3 PROPOSED DESIGN FLOW ENGINE

This section describes the design flow engine which has been proposed for the description and the execution of complex and customized High Level Synthesis design flows. Its main novelty is that the passes which compose it can dynamically modify and tune the current design flow without requiring a full evaluation of intermediate solutions nor time-consuming design space exploration. In this way, High Level Synthesis flow can be customized during its execution on the basis of the outcomes of already executed passes. This result cannot be achieved with pass managers based on statically fixed sequences of passes (e.g., the pass managers currently implemented in GCC and LLVM), even if they contain repetitions, nor with hierarchical design flows.

To achieve this objective, the proposed approach includes the following characteristics:

- *Evolving graph*: it represents the currently considered design flow; design flow passes can be added, re-executed, or disabled, depending on the outcomes of already executed passes; similarly, dependencies and precedences among passes can be added and removed.
- *Dynamic scheduling*: in each moment the next pass to be executed can be deterministically computed on the basis of the information provided by all the passes.
- *Composition of multiple design flows*: the overall design flow can be obtained as the combination of multiple specialized design flows.
- *Cyclic dependencies*: there can be cyclic dependencies among passes: a pass can invalidate the execution of one or more previously executed passes requiring its re-execution, even if they both transform intermediate representations.

With respect to pass managers adopted in traditional compilation flows, the proposed infrastructure supports a much wider possibility of implementing complex design flows, but it can increase the overall design time because of its larger complexity. However, in design scenarios like High Level Synthesis, this overhead is fully affordable, provided that the more complex design flows can introduce a significant benefit in terms of quality of results.

In the following, Section 3.1 details the *Graph of Passes*, which describes the current status of the design flow, while Section 3.2 shows how this graph is updated every time the execution of a pass ends. Finally, Section 3.3 presents how the proposed approach can be integrated into existing compilers and High Level Synthesis tools.

### 3.1 Graph of Passes

The design flow is modeled as a directed graph called *Graph of Passes* whose vertices are the passes that have to be executed and whose edges are the precedences and dependencies that have to be preserved during the execution. A single flat graph is used: hierarchical design flows can be described, but then the corresponding graphs will be flattened in a single *Graph of Passes*. A different vertex is added for each pass and for each separate portion on which it is applied (e.g., a different instance of a function analysis pass is created for each function). On the contrary, the different executions of the same pass on the same (modified) portion of intermediate representation are represented by a single vertex. There is not any distinction among analysis passes and optimization

Table 1. The different types of edges of *Graph of Passes*.

| | |
|---|---|
| *Prerequisite* | A *Prerequisite* edge $A \longrightarrow B$ is added if pass $B$ requires the previous execution of $A$. |
| *Precedence* | A *Precedence* edge $A \longrightarrow B$ is added if pass $A$ cannot be executed after $B$; this, differently from *Prerequisite*, does not imply that $B$ requires the execution of $A$ so that $A$ has to be executed only if it is also a prerequisite of some another pass. |
| *Invalidation* | An *Invalidation* edge $A \longrightarrow B$ is added if the last execution of pass $A$ invalidates the execution of $B$ requiring executing again $B$. If the last execution of $A$ does not invalidate pass $B$ and the edge was previously added, this is removed. |

passes, but two special vertices (*Entry* and *Exit*) are added to represent the beginning and the end of the design process.

The *Graph of Passes* has three different types of edges which describe the possible relationships between pairs of passes and which are listed in Table 1. All the passes are assumed to have *Entry* as a prerequisite and to be precedences of *Exit*. If a pass is a *Precedence* of another, the corresponding vertex is added to the *Graph of Passes* as a placeholder to correctly manage possible future dependencies, but it will not be executed if there is not any other pass which requires it as a prerequisite. Differently from other proposed design flow, a pass can invalidate passes of whatever type. The choice about when a pass must invalidate some of its predecessors is delegated to the pass itself and should be based on changes of the intermediate representations. Examples of changes which can trigger passes invalidation are the transformations applied on IRs (e.g., CSE, code motion), the availability of more detailed information (e.g., tighter variable ranges induced by improved bit-value analysis, new memory allocation due to memory accesses refinements), and the introduction of new functions (e.g., introduction of memcpy to implement memory copy). In the case of invalidation of analysis passes, this corresponds to invalidating the results of the corresponding analyses. In the case of invalidation of optimization passes, this corresponds to executing again the optimization starting from the current version of the intermediate representation. In this way, it is possible to describe complex and customized design flows based on fixed-point optimizations. For example, if three optimizations have to be executed in sequence until the intermediate representation does not change anymore (i.e., fixed-point optimization flow), the last optimization pass will continue to invalidate the first until there will not be any new change in the intermediate representation. Similarly, if the early optimizations can benefit from the last passes, the latter can invalidate the former provoking their re-execution. *Invalidation* edges are the only edges of the graph which can close a cyclic path (i.e., they are the only feedback edges) making the introduction of cyclic dependencies explicit.

There is a special class of passes which do not implement any analysis nor optimization but are only characterized by a set of dependencies and prerequisites. This type of passes, called *Composing passes*, can be exploited to hide to other passes how a part of the design flow is implemented making the management of dependencies easier. There are three main aims for which they can be used:

- To represent a flow composed of several passes as a single prerequisite; for example, a *Composing pass* can represent the whole flow of intermediate representation transformations if it has all of them as prerequisites, while their order of execution will be computed on the basis of the prerequisites and precedences specified by the single passes. The passes which must be executed after all the intermediate representation transformations (e.g., actual High Level Synthesis passes like scheduling and module binding) can have the *Composing pass*

Table 2. The possible statuses of a pass in *Graph of Passes*.

| | |
|---|---|
| *Non Existent* | The pass does not yet exist in the graph. |
| *Success* | The pass has already been executed and it has changed the intermediate representation. |
| *Unchanged* | The pass has already been executed and it has not changed the intermediate representation. |
| *To Be Executed* | The pass must be executed (or re-executed because of invalidation). It has been included as *Prerequisite* of another *To Be Executed* pass. |
| *Unnecessary* | No pass requires its execution, but its successors cannot yet be executed. It can be included because it is a *Precedence* of any pass or a *Prerequisite* of another *Unnecessary* pass. |
| *Skipped* | No pass requires its execution and its successors can be executed. It corresponds to an “executed” *Unnecessary* pass. |

as the only prerequisite pass. In this way, the details about which are the transformations performed on the intermediate representation can be hidden to the rest of the passes of the design flow.

- To represent the same pass applied to different portions of the intermediate representation. For example, a *Composing pass* can represent the application of data flow analysis to all the functions of the specification to be synthesized; this pass has as prerequisites an instance of the data flow analysis pass for each function of the specification.
- To represent an analysis or optimization pass independently from the implemented algorithm. For example, the code motion pass depends on the alias analysis pass, but alias analysis can be implemented with different algorithms. This prerequisite can be specified as follows: a *Composing pass* is added representing a generic alias analysis pass, which has as a prerequisite the actual pass implementing the alias analysis. This can be selected during the design process, for example, on the basis of the current status of the intermediate representation or on the basis of the options provided by the user. Finally, the alias analysis *Composing pass* must be added as a prerequisite of the code motion pass.

Note that differently from other previous approaches, the presented design flow engine does not preserve the hierarchy among flows, but flattens all the design flows in a single graph. While considering at the same time all the passes of the design flow has the disadvantage of incrementing the complexity and the overhead management time of the *Graph of Passes*, this allows the designer to specify fine-grain relationships among passes. For example, the flattened design flow allows a back-end pass to invalidate only partially the middle-end flow.

Each vertex of the graph also describes the status of the pass associated with it, which is a piece of significant information since it contributes to the selection of the next pass to be executed. Correctly choosing it is critical since useless executions must be prevented as well as infinite repetitions. The statuses which a pass can have are listed in Table 2. In the next section, it will be shown how the status of the single passes is updated during the design process.

## 3.2 Scheduling and Update of the Graph of Passes

Since the *Graph of Passes* evolves during the design flow (i.e., new passes and relationships are added), the scheduling of the passes cannot be computed globally once at the beginning of the execution of the design flow. On the contrary, each time a pass ends, which is the next pass to be executed must be computed, since new passes can become ready and passes which were ready can

**ALGORITHM 1:** Design

```
input : design_flow_type
data  : graph
data  : ready_passes
1  RecAddPasses(design_flow_type, graph, true)
2  ready_passes ⟵ {Entry }
3  while Not ready_passes.Empty() do
4      candidate_pass ⟵ GetReadyPass()
5      current_status ⟵ candidate_pass.GetStatus()
6      RecAddPasses(candidate_pass, graph, current_status == To Be Executed)
7      ready_passes ⟵ UpdateReadyPasses()
8      if candidate_pass ∉ ready_passes then
9          Continue
10     end
11     if current_status == Unnecessary then
12         candidate_pass.SetStatus(Skipped)
13     end
14     else
15         current_status ⟵ candidate_pass.Exec()
16         candidate_pass.SetStatus(current_status)
17     end
18     passes ⟵ candidate_pass.GetInvalidations()
19     foreach pass ∈ passes do
20         graph.InvalidatePass(pass)
21     end
22     ready_passes ⟵ UpdateReadyPasses()
23 end
```

be not ready anymore since they have new requirements. Which is the next pass to be executed is computed by means of Algorithm 1. Its input is the type of High Level Synthesis flow to be executed which is represented by a *Composed* pass. The algorithm recursively computes its precedences and its prerequisites (line 1) which mainly consists of other *Composed* passes describing the different phases of the design flow; the details of this part of the computation are described by Algorithm 2. The set of passes which are added during this initialization is a very limited subset of the overall set of passes which will be added during the design flow execution. At the beginning of the design flow, for example, it is not possible to know which are the functions which compose the specification and it is not possible to add any function analysis or function optimization pass.

The only pass whose execution can be ready at the beginning is the *Entry* pass (line 2): all the other passes have at least it as a prerequisite. Next, the iterative execution of the design flow starts and continues while there are some ready passes to be executed: the last pass to be executed is the *Exit* pass. During each iteration (lines 3-23) the *Graph of Passes* is updated as follows:

(1) The pass candidate to be executed next is deterministically selected (line 4) among the ready passes; a pass is ready if all its predecessor have been executed (i.e., they have one of the following statuses: *Success*, *Unchanged*, *Skipped*).
(2) The precedences and the prerequisite of the candidate pass are recomputed (line 6) by re-adding the pass. Since they can depend on the status of the design flow and the outcome of the previous passes, they can be different with respect to their previous computation.

**ALGORITHM 2:** RecAddPasses

```
input : inpass
input : graph
input : to_be_executed
1  if inpass.GetStatus() == Skipped then
2      graph.InvalidatePass(inpass)
3  end
4  graph.AddPass(inpass, to_be_executed)
5  passes ← inpass.GetPrecedences()
6  foreach pass ∈ passes do
7      RecAddPasses(pass, graph, false)
8  end
9  passes ← inpass.GetPrerequisites()
10 foreach pass ∈ passes do
11     RecAddPasses(pass, graph, to_be_executed)
12 end
```

(3) The set of ready passes is recomputed (line 7) since if the candidate pass has new unsatisfied precedences or prerequisites it is not ready anymore.
(4) If the candidate pass is not anymore ready (line 8), the current iteration is aborted and a new candidate will be selected.
(5) If the candidate pass is still ready, there are two possible scenarios:
   - Its status is *Unnecessary* (line 11): its status is updated to *Skipped* (line 12) potentially allowing the execution of its successors (i.e., a successor becomes ready if the current pass was the only *To Be Executed* or *Unnecessary* predecessor).
   - Its status is *To Be Executed* (line 14): the pass is executed (line 15) and its status is updated (line 16).
(6) The candidate pass is queried to retrieve the set of passes which it invalidates if any. Their direct and indirect successors are also invalidated (lines 18-21).
(7) Finally, the set of ready passes is updated (line 22).

Algorithm 2 shows the details of the recursive computation of the precedences and the prerequisites of a given pass. The algorithm takes as input the pass whose relationships have to be computed, the *Graph of Passes*, and if it is being added as a *To Be Executed* or as an *Unnecessary* pass. If the pass already exists in the graph and it has *Skipped* status (line 1), Then all its successors must be invalidated since they worked on an old version of the intermediate representation (line 2). Then, the pass is added to the graph (or its status is updated according to the rule of Table 3) through `AddPass` (line 4). Next its precedences are added (lines 5-8) as *Unnecessary* passes, and finally, its prerequisites are added (lines 9-12). It is worth noting that the set of prerequisites and precedences of a pass may depend on the current status of the design flow and in particular on all the analyses and optimizations which have been applied so far. For this reason, a global schedule of all the passes to be executed cannot be pre-computed at the beginning of the execution of the design flow. Note that given three passes $A$, $B$, $C$ and provided that only the execution of $C$ is required:

- if $A$ is a precedence of $B$ and $B$ is a prerequisite of $C$, $A$ is *Unnecessary*, $B$ is *To Be Executed*,
- if $A$ is a precedence of $B$ and $B$ is a precedence of $C$, $A$ and $B$ are *Unnecessary*.
- if $A$ is a prerequisite of $B$ and $B$ is a prerequisite of $C$, $A$ and $B$ are *To Be Executed*.
- if $A$ is a prerequisite of $B$ and $B$ is a precedence of $C$, $A$ and $B$ are *Unnecessary*.

Table 3. The new status of a pass after the invocation of AddPass and InvalidatePass.

| | **New Status** of *pass* after | | |
|---|---|---|---|
| **Initial Status** of *pass* | AddPass(*pass*, true) | AddPass(*pass*, false) | InvalidatePass(*pass*) |
| *Non-existent* | *To Be Executed* | *Unnecessary* | – |
| *Unnecessary* | *To Be Executed* | *Unnecessary* | – |
| *Success* | *Success* | *Success* | *To Be Executed* |
| *To Be Executed* | *To Be Executed* | *To Be Executed* | – |
| *Unchanged* | *To Be Executed* | *To Be Executed* | *To Be Executed* |
| *Skipped* | *To Be Executed* | *Skipped* | *Unnecessary* |

Because of the last combination, it is possible that the status of a prerequisite of a pass is *Unnecessary* (e.g., *A* is a prerequisite of *B*, but it has not to be executed).

`InvalidatePass(pass)`, used by Algorithm 1 and Algorithm 2, invalidates a pass and recursively all its successors (direct or indirect) in the *Graph of Passes*. The pass which is directly invalidated must have the status of *Success*, *Unchanged*, or *Skipped*; the passes which are indirectly invalidated by the recursion are updated according to the rules shown in the right part of Table 3.

The deterministic selection of the candidate pass makes the whole design flow deterministic (provided that the single passes are deterministic too). The absence of loops composed only of *Prerequisite* and *Precedence* edges guarantees that the recursive call of `RecAddPasses` ends and that there is always at least one ready pass which can be executed. Nevertheless, these conditions do not guarantee that the design flow ends: a wrong invalidation, for example, which is introduced independently from the results of the current pass, generates a cyclic dependence in the *Graph of Passes* that, since it will never be removed, causes the non-termination of the design flow. The introduction of an upper bound of the maximum number of executions of each pass, however, can mitigate this issue. Moreover, it has to be noted that the termination problem is not necessarily introduced in a design flow by the inclusion of potentially circular dependencies among passes. Indeed, even a linear design flow does not terminate if at least one of the passes which compose it does not terminate.

## 3.3 Integration in existing compilers and High Level Synthesis tools

The proposed approach has been implemented in our High Level Synthesis tool as described in Section 5.1. However, since the proposed solution is general and does not significantly rely on a particular infrastructure, this can be integrated also in other High Level Synthesis tools and more in general in other existing compilers such as LLVM. To achieve this objective, three steps are required. The first step consists of replacing the existing pass manager with the implementation of the proposed solution (i.e, the *Graph of Passes*, the functions for its update, and the functions to select the next pass to be executed). Since most of the state of the art compiler-based infrastructures (e.g., GCC, LLVM) already include a pass manager, this is straightforward. The second step consists of modifying the common interface of all the passes (if already exist like in LLVM, or introducing it if does not exist) by adding the functions which return the set of prerequisites, of precedences, and of invalidated passes for each available pass. In the case of LLVM, information about the required and invalid analyses has to be included in these functions. Moreover, the optimization pass which precedes the current in the static design flow has to be added as a prerequisite.

Applying these first two steps is quite easy, but they do not provide significant benefits since the same behavior of a static pass manager is obtained. To exploit the advanced features of the proposed approach the third step, which potentially is the most complex, must be performed. This step consists of enriching the functions for computing the relationships among passes, for example, by adding invalidation of passes when their repetition can be profitable, and of modifying the passes to support the new potential design flows. How this can be obtained is something strictly related to the considered pass. There are indeed three main reasons for which passes must be partially redesigned to fully exploit the functionalities provided by the proposed approach:

- The passes must support multiple re-execution. This may require to introduce the possibility of re-initialize them deleting all the already computed data.
- Changing the order of the execution of the passes can generate patterns in the intermediate representation which were not originally foreseen, preventing possible further optimizations or provoking the abort of the execution of a pass.
- Invalidation of already executed passes can be profitable provided that the invalidated passes can exploit the extra computed information.

## 4 CASE STUDY

This section presents an example of complex High Level Synthesis flow which can be described and executed using the design flow engine proposed in this paper. For the sake of brevity and readability, only some of its most significant parts will be highlighted and only some of its passes have been included in Figure 1 which summarizes it. Moreover, a transitive reduction has been applied to the graph to reduce the number of shown edges. Roughly, a typical High Level Synthesis flow such as the one presented in this case study is composed of a *C Parser* pass, which generates the initial intermediate representation, several *Middle-End Analyses and Optimizations* passes (e.g., *Local Code Motion, Empty BB Removal*), which manipulate the intermediate representation to make it more suitable to be synthesized, some *High Level Synthesis* passes (e.g., *Memories Allocation, SDC Scheduling*), which actually generate the circuit, and, finally, a backend generation pass (i.e., *HDL Generation*).

The list of the passes to be executed in the presented design flow is not statically computed, but it is dynamically updated at the end of the execution of each pass. At the beginning, the *Graph of Passes* is composed only of *Synthesis Flow*. Then, by recursively adding its precedences and prerequisites, the composed passes representing the different sub-flows (i.e., *Finalize IR* and *High Level Synthesis*) and the writer (i.e., *HDL Generation*) are added to the graph. Next, the prerequisites of composed passes are added. In particular, only the passes which work on the whole specification (e.g., *Call Graph Computation*, *Functions Allocation*) can be added. On the contrary, the passes which refer to a single function cannot be added since the set of functions composing the specification is not yet known. Note that in Figure 1 some edges, which have been added during the evolution of the design flow (e.g., the edge from *Memories Allocation* to *High Level Synthesis*), are not shown because of the applied transitive reduction. The numbers in the left part of the vertices of Figure 1 are the order of execution of the passes dynamically computed by the design flow engine. Strikethrough numbers identify the pass executions which are skipped because the necessary information is not yet available (e.g., *SDC Code Motion (main)* at 4), because the input intermediate representation does not change with respect to the previous execution (e.g., *Conditions Merging (func1)* at 10), or because they reached their maximum execution number (e.g., *SDC Scheduling (main)* at 47). This order cannot be statically computed since it depends on the invalidations required by *Conditions Merging* and by *SDC Scheduling* which depend on their outcome.

The first steps to be executed are the parsing of the C code and the successive building of the call graph. Two functions are identified: `func1` and `main`, so all the required middle-end passes (B) are added to the graph as direct and indirect prerequisites of *Finalize IR*. Note that inter-procedural

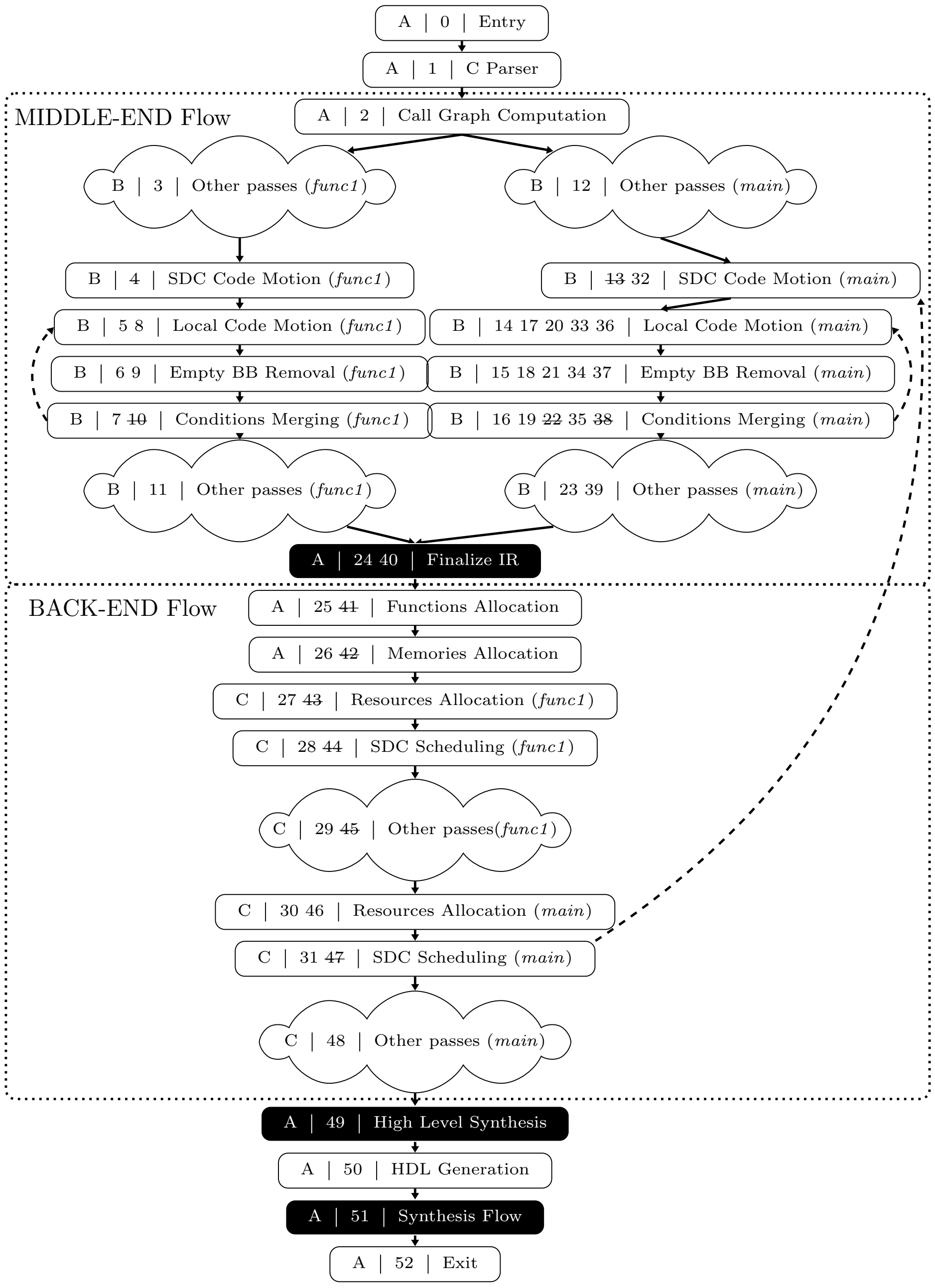

A | 0 | Entry
A | 1 | C Parser
MIDDLE-END Flow
A | 2 | Call Graph Computation
B | 3 | Other passes (func1)
B | 12 | Other passes (main)
B | 4 | SDC Code Motion (func1)
B | ~~13~~ 32 | SDC Code Motion (main)
B | 5 8 | Local Code Motion (func1)
B | 14 17 20 33 36 | Local Code Motion (main)
B | 6 9 | Empty BB Removal (func1)
B | 15 18 21 34 37 | Empty BB Removal (main)
B | 7 ~~10~~ | Conditions Merging (func1)
B | 16 19 ~~22~~ 35 ~~38~~ | Conditions Merging (main)
B | 11 | Other passes (func1)
B | 23 39 | Other passes (main)
A | 24 40 | Finalize IR
BACK-END Flow
A | 25 ~~41~~ | Functions Allocation
A | 26 ~~42~~ | Memories Allocation
C | 27 ~~43~~ | Resources Allocation (func1)
C | 28 ~~44~~ | SDC Scheduling (func1)
C | 29 ~~45~~ | Other passes(func1)
C | 30 46 | Resources Allocation (main)
C | 31 ~~47~~ | SDC Scheduling (main)
C | 48 | Other passes (main)
A | 49 | High Level Synthesis
A | 50 | HDL Generation
A | 51 | Synthesis Flow
A | 52 | Exit

Fig. 1. Example of Design Flow Graph implementing the High Level Synthesis of a specification composed of two functions (func1 and main). Black boxes are *Composing passes*. The letters identify the order of insertion of the passes, the numbers identify the order of execution of the passes, the canceled numbers represent skipped passes while the dashed edges represent the invalidations. A passes are added at the beginning, B passes after iteration 2, C passes after iteration 26. For the sake of readability, a transitive reduction was applied.

analysis passes are fully supported by the presented design flow engine as well as passes which can be executed only after the analysis of the called or calling functions.

The middle-end flow is not linear: sequences of passes can be executed multiple times and the number of executions can be different according to the particular function. An example of repeated optimizations consists of *Local Code Motion*, *Empty BB Removal*, and *Conditions Merging*. These optimization aim at restructuring the Control Flow Graph to optimize the latency in terms of the number of corresponding clock cycles of all its paths. *Local Code Motion* tries to empty basic blocks by moving their instructions in other basic blocks. To guarantee that these movements do not increase the latency of any path, the pass must guarantee that adding an instruction to a basic block does not increase its latency. However, since the available information about the latency of the instructions is very limited (most of the IR optimizations and the whole High Level Synthesis flow have not yet been executed), only very conservative transformations can be applied. For example, moving zero delay operations is safe and can be always performed. To gain benefits from this code motion, the empty basic blocks, if any, must be removed, and this is performed in the next pass. As a result, sequences of basic blocks composed only of conditional constructs may appear. These can be further optimized by merging their conditions. Since the last pass can create new instructions which define the more complex conditions, new opportunities of code motion may arise, so that it can be profitable to re-execute *Local Code Motion*. For this reason, this is invalidated as well as *Empty BB Removal* and *Conditions Merging*. The whole process is repeated until there is not any transformation which can be applied. Precompute how many times this cycle has to be repeated by performing a simple analysis of the specification is not possible since previous passes can already modify the structure of the Control Flow Graph. Moreover, the obtained solution is not necessarily the best one, since more information would allow the pass to perform other local code motions, but detailed information will be available only later in the flow. Note that these transformations are different from the passes implemented in a compiler targeting general-purpose processors because of the different characteristics of an FPGA. Code motion transformations indeed can be more aggressive since on an FPGA it is possible to execute in parallel an arbitrary number of operations. Moreover, it is possible to create zero-delay branches with an arbitrary number of conditions and target destinations.

After the optimization of the intermediate representation, the actual High Level Synthesis can start applying some global passes (i.e., *Functions Allocation* and *Memories Allocation*). At this point, the only read pass, which is the candidate to be executed, is *High Level Synthesis*. Before executing it, however, its direct and indirect prerequisites are recomputed causing the inclusion of other back-end passes (C) to the design flow. Note that before the execution of *Call Graph Computation* these prerequisites were not present. The order of synthesis of functions is not arbitrary: called functions must be synthesized before calling functions, so `func1` is synthesized before `main`.

One of the most significant passes to be applied is the *SDC Scheduling* [16]: this pass computes a global schedule of the whole function and then suggests some code motions to be applied to improve the performance which are applied by the *SDC Code Motion* pass. For some simple functions, or in general for functions composed of a single basic block, *SDC Scheduling* does not suggest any code motion. In this case, like for `func1` of the example, it is not necessary to invalidate *SDC Code Motion* (its execution would not change the intermediate representation) nor any of the other passes of the middle-end flow. For this reason, even after the first execution of *SDC Scheduling (func1)*, the flow continues with the rest of High Level Synthesis. On the contrary, the moving of some instructions is suggested by *SDC Scheduling* in case of `main` function, resulting in the invalidation of *SDC Code Motion (main)*. Differently from *Local Code Motion*, the *SDC Code Motion* pass allows movements of operations which can increment the delay of some basic blocks provided that the overall delay of each path of the Control Flow Graph is not incremented. Such type of information can be computed only by *SDC Scheduling* when all the characteristics of the single instructions are available. For example, it is necessary to know for each instruction which is the size of its operands, on which type of functional unit it has been mapped, where its inputs and output are stored, and so on. Foreseeing and estimating all this information before the starting of the actual High Level Synthesis flow will introduce very significant inaccuracies, potentially causing unprofitable movements.

Exploiting *SDC Code Motion* in a linear flow as a pass inserted between the *SDC Scheduling* and the successive passes is not affordable for two reasons. The first one is that the intermediate representation has been modified, so the High Level Synthesis flow cannot continue from the current pass, but some passes have to be re-executed. For example, since some instructions have been moved, the liveness of the variables used by them may change requiring to recompute this information. The second reason is that the solution suggested by SDC scheduling cannot be fully implemented through code movings resulting in generating an intermediate representation that can be still improved by middle-end optimization passes.

To force the re-execution of some of the middle-end passes after *SDC Code Motion*, this has been added as a prerequisite of *Local Code Motion*. The invalidation of the former invalidates the latter and all the successive passes. Moreover, optimization passes not only can still change the intermediate representation since they work on a different version of it, but they can be applied more effectively because of the enriched information. For example, *Local Code Motion* during its first execution can perform only very conservative moves since it does not know the delay of the single instructions nor if there are enough resources to execute in parallel all the operations of a given basic block. On the contrary, during its second execution, since the whole allocation information is now available, the scheduling of the operations of each basic block can be computed allowing the pass to apply more aggressive code motion. Other examples of passes which can exploit High Level Synthesis information are bit-width analysis and optimization. *Memories Allocation* computes information about the size of the bus addresses: this information can be exploited for resizing the width of the operands of all the operations related to pointer arithmetic, potentially reducing the size of the generated accelerator and potentially exposing further opportunities of optimizations.

Since the synthesis of a function depends on the synthesis of the called functions, it is necessary to repeat multiple times the loop composed of passes of the middle-end and passes of the back-end. However, if the used intermediate representation remains the same, the passes can exploit the already computed information to avoid their re-execution, significantly reducing the number of passes which are re-executed and consequently the overall synthesis time. For example, the middle-end passes of the `func1` functions have not been invalidated, so they have not to be re-executed. The High Level Synthesis passes of the same function instead have been invalidated, but their execution can also be skipped since function intermediate representation does not change since their last execution. On the contrary, *Resources Allocation (main)* has to be executed two times because of the changes in the intermediate representation between the two executions. The adopted solution allows a High Level Synthesis pass to invalidate a subset of the Middle-End optimizations instead of invalidating all the passes. This fine-tuning of relationships is possible only with a flattened graph and not with a hierarchical graph.

## 5 EXPERIMENTAL EVALUATION

This section shows the benefits provided by the exploitation of a dynamic design flow with respect to static design flow. In particular, Section 5.1 presents the implementation of the proposed design flow engine in Bambu and the considered experimental setup, while Section 5.2 presents the obtained results.

### 5.1 Experimental Setup

The proposed design flow engine has been implemented in Bambu [20], an open-source publicly available[1] tool for High Level Synthesis which is part of the PandA framework. The aim of Bambu is the automatic generation of hardware circuit implementations starting from C source code specifications. The tool uses as C frontend the GNU GCC through its plugin API which is exploited to extract the GIMPLE intermediate representation. The tool does not rely on a particular version of the compiler: all the versions since GNU GCC 4.5 are currently supported. The adoption of GNU GCC as front end removes any issue due to the parsing of C source code and avoids reimplementing all the state-of-the-art optimization techniques which are already implemented in GNU GCC since the intermediate representation is extracted after the middle-end optimizations. The current version of Bambu supports the synthesis of most of the constructs of the C language: the main construct which is not supported is the recursive calls. Moreover, Bambu supports the synthesis of specifications described by multiple C source code files: for each of them, GNU GCC is called making it produce the corresponding GIMPLE intermediate representation. All the generated intermediate representations are then collected and merged by a pseudo linker pass implemented in the PandA framework. The actual High Level Synthesis flow is not immediately applied to the outcome of the GNU GCC: a set of analyses and optimizations are indeed applied on the GIMPLE-like intermediate representation before High Level Synthesis. Indeed, since FPGAs are a target very different from general-purpose processors, there is a set of intermediate representation optimizations which are profitable only for this type of devices and that, for this reason, are not implemented in GNU GCC. Some examples of these optimizations are detailed bitsize analysis, expansion of multiplications by constants, and aggressive code motion transformation. Next, the actual High Level Synthesis flow can start and finally the optional last passes of the implemented design flow can be executed. These are the simulation of the generated design for the evaluation of its performance and its verification [8], and the generation of the logic gates architecture, which will be implemented on the FPGA device, performed by external Logic Synthesis tools [17].

Three different High Level Synthesis flows have been considered:

- *Sequential*: it is a High Level Synthesis flow without any cyclic dependencies among passes (i.e., without invalidations).
- *Default*: it is the default High Level Synthesis flow implemented in Bambu; it includes local cyclic dependencies.
- *SDC*: it is the High Level Synthesis flow [16] based on SDC scheduling [6] with cyclic dependencies between Middle-End Optimization passes and High Level Synthesis passes.

It is worth noting that both the *Default* and the *SDC* High Level Synthesis flows requires the dynamic update of sequences of passes provided by the proposed infrastructure. The same result cannot be obtained by exploiting state-of-the-art pass managers (e.g., the ones integrated into GCC or LLVM) which are based on linear fixed design flows nor by exploiting only hierarchical design flows.

To prove the generality of the proposed approach two different scenarios have been considered. In the first scenario, the considered target is the Xilinx Zynq-7000 xc7z0 with a target frequency of 66.66 *MHz*. The final synthesis from the RTL description to the bitstream is performed with Xilinx Vivado [25]. The three design flows have been applied to the benchmarks of the CHStone suite [11], which aim at representing all the possible scenarios which have to be addressed by a High Level Synthesis tool. The obtained results have been compared with the results obtained with a commercial tool. In the second scenario, the considered target is the Stratix V 5SGXEA7N2F45C1

[1]https://github.com/ferrandi/PandA-bambu

Table 4. Characteristics of the three design flows when targeting the Zynq board. *|V|*: the size of the *Graph of Passes* in terms of the number of vertices. *|E|*: the size of the *Graph of Passes* in terms of the number of edges. *Exec*: the overall number of executions of passes. *Skip*: the overall number of executions of passes that are skipped since not necessary. *Mods*: the overall number of times the *Graph of Passes* is updated. *PG time*: the execution time spent in updating the *Graph of Passes* and selecting the next pass to be executed. *HLS time*: the execution time of the design flow implemented in Bambu. *LS Time*: the execution time required by Logic Synthesis and Place-and-Route.

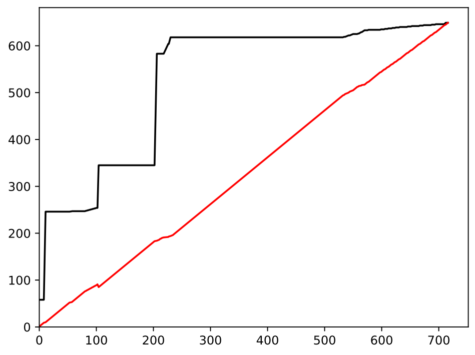
| | | Final Size of Graph of Passes | | Number of Passes | | | Execution Times (Seconds) | | |
|---|---|---|---|---|---|---|---|---|---|
| | Flow | \|V\| | \|E\| | Exec | Skip | Mods | PG | HLS | LS |
| adpcm | *Sequential* | 151 | 629 | 138 | 3 | 18 | 0.00 | 10.91 | 724.81 |
| | *Default* | 151 | 635 | 383 | 49 | 26 | 0.02 | 10.88 | 538.41 |
| | *SDC* | 155 | 656 | 589 | 113 | 32 | 0.07 | 38.46 | 587.48 |
| aes | *Sequential* | 483 | 2229 | 442 | 15 | 38 | 0.05 | 41.66 | 400.56 |
| | *Default* | 483 | 2253 | 952 | 561 | 57 | 0.14 | 38.42 | 263.99 |
| | *SDC* | 495 | 2329 | 1708 | 1883 | 74 | 0.39 | 89.18 | 338.96 |
| blowfish | *Sequential* | 234 | 1029 | 214 | 6 | 23 | 0.02 | 17.32 | 247.55 |
| | *Default* | 234 | 1037 | 505 | 187 | 34 | 0.05 | 8.49 | 198.62 |
| | *SDC* | 240 | 1071 | 816 | 401 | 42 | 0.11 | 26.37 | 233.00 |
| dfadd | *Sequential* | 151 | 629 | 138 | 3 | 18 | 0.00 | 3.78 | 299.93 |
| | *Default* | 151 | 635 | 546 | 54 | 32 | 0.04 | 14.67 | 164.87 |
| | *SDC* | 155 | 656 | 1235 | 124 | 53 | 0.11 | 78.26 | 212.66 |
| dfdiv | *Sequential* | 234 | 1024 | 213 | 6 | 25 | 0.03 | 3.86 | 419.67 |
| | *Default* | 234 | 1036 | 897 | 309 | 48 | 0.04 | 9.11 | 260.45 |
| | *SDC* | 240 | 1070 | 1384 | 596 | 61 | 0.19 | 25.50 | 355.14 |
| dfmul | *Sequential* | 151 | 629 | 138 | 3 | 18 | 0.02 | 3.14 | 228.51 |
| | *Default* | 151 | 635 | 541 | 58 | 32 | 0.02 | 6.03 | 141.28 |
| | *SDC* | 155 | 656 | 861 | 109 | 41 | 0.06 | 18.89 | 203.16 |
| dfsin | *Sequential* | 317 | 1424 | 289 | 9 | 30 | 0.04 | 10.35 | 866.85 |
| | *Default* | 317 | 1442 | 1005 | 461 | 55 | 0.12 | 30.32 | 549.21 |
| | *SDC* | 325 | 1490 | 1918 | 1338 | 78 | 0.35 | 187.30 | 685.91 |
| gsm | *Sequential* | 234 | 1029 | 214 | 6 | 23 | 0.01 | 5.86 | 461.75 |
| | *Default* | 234 | 1041 | 749 | 216 | 41 | 0.05 | 6.22 | 297.97 |
| | *SDC* | 240 | 1076 | 986 | 411 | 46 | 0.10 | 18.11 | 373.93 |
| jpeg | *Sequential* | 566 | 2639 | 518 | 18 | 43 | 0.06 | 23.52 | 727.98 |
| | *Default* | 566 | 2673 | 1555 | 1530 | 80 | 0.31 | 16.68 | 732.00 |
| | *SDC* | 580 | 2769 | 2602 | 3913 | 105 | 0.98 | 45.12 | 648.49 |
| mips | *Sequential* | 151 | 629 | 138 | 3 | 18 | 0.01 | 3.39 | 215.52 |
| | *Default* | 151 | 635 | 331 | 42 | 24 | 0.02 | 3.86 | 150.09 |
| | *SDC* | 155 | 657 | 458 | 80 | 29 | 0.04 | 20.84 | 194.51 |
| mpeg2 | *Sequential* | 400 | 1834 | 366 | 12 | 33 | 0.03 | 4.14 | 202.63 |
| | *Default* | 400 | 1854 | 831 | 379 | 50 | 0.08 | 2.31 | 155.39 |
| | *SDC* | 410 | 1915 | 1436 | 1402 | 64 | 0.31 | 5.20 | 224.54 |
| sha | *Sequential* | 234 | 1029 | 214 | 6 | 23 | 0.01 | 4.35 | 219.41 |
| | *Default* | 234 | 1039 | 478 | 147 | 33 | 0.10 | 6.50 | 173.28 |
| | *SDC* | 240 | 1075 | 875 | 407 | 43 | 0.17 | 10.36 | 225.82 |

from Intel, while the set of considered benchmarks is the same of [18]. This set also contains the *CHStone* benchmarks, even if with different setup (i.e., different functions to be synthesized). Logic Synthesis and Place-and-Route are executed using Intel Quartus Pro and the results have been compared with the results of LegUp for the unopt setup [18]. For each benchmark, the selected target frequency is the one achieved by LegUp.

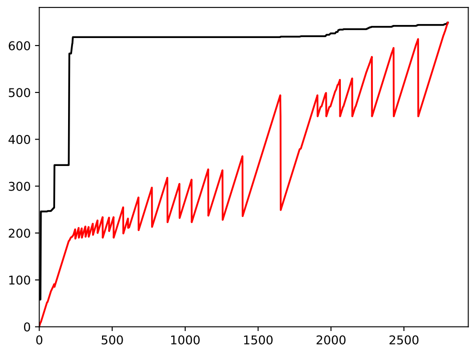
In both the scenarios, all the experiments have been run on an Intel E5-2620 running at 2.10GHz with 32GB of memory.

## 5.2 Experimental Results

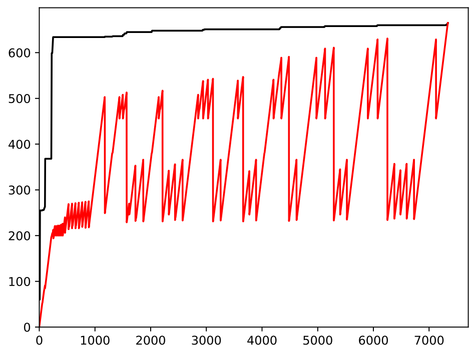
Table 4 reports the size of each type of High Level Synthesis flow in the first scenario. For each combination of benchmark and design flow the table reports:



(a) *Sequential* design flow



(b) *Default* design flow



(c) *SDC* design flow

Fig. 2. Size of *Graph of Passes* in terms of the overall number of vertices (black lines) and in terms of the sum of passes marked as *Success*, *Unchanged*, and *Skipped* (red lines) during the execution of design flows for the synthesis of motion benchmark targeting Intel Stratix V.

Table 5. Characteristics of the solutions obtained by the commercial tool and by the three design flows when targeting Zynq.

| | Flow | Cycles | Freq. [MHz] | Wall-time [$\mu s$] | LUTs | Registers | BRAMs | DSPs |
|---|---|---|---|---|---|---|---|---|
| adpcm_encode | *Commercial Tool* | 23075 | 34.30 | 672.74 | 10358 | 3488 | 13 | 46 |
| | *Sequential* | 17128 | 66.75 | 256.61 | 9332 | 6184 | 16 | 70 |
| | *Default* | 16582 | 67.17 | 246.87 | 9185 | 6334 | 10 | 43 |
| | *SDC* | 15402 | 73.66 | 209.10 | 9166 | 5640 | 10 | 37 |
| aes | *Commercial Tool* | 2973 | 89.20 | 33.32 | 3625 | 1888 | 13 | 6 |
| | *Sequential* | 2729 | 80.65 | 33.84 | 4681 | 2446 | 8 | 0 |
| | *Default* | 2601 | 79.45 | 32.74 | 4527 | 2063 | 8 | 0 |
| | *SDC* | 2592 | 78.44 | 33.04 | 4738 | 2444 | 8 | 0 |
| blowfish | *Commercial Tool* | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| | *Sequential* | 93954 | 90.22 | 1041.39 | 2716 | 2025 | 20 | 0 |
| | *Default* | 96664 | 86.51 | 1117.34 | 2649 | 1967 | 20 | 0 |
| | *SDC* | 96534 | 91.47 | 1055.31 | 2644 | 1955 | 20 | 0 |
| dfadd | *Commercial Tool* | 383 | 72.00 | 5.32 | 2970 | 1544 | 0 | 0 |
| | *Sequential* | 537 | 79.78 | 6.73 | 2887 | 2043 | 0 | 0 |
| | *Default* | 293 | 78.82 | 3.72 | 2359 | 1433 | 0 | 0 |
| | *SDC* | 249 | 74.85 | 3.33 | 2108 | 924 | 0 | 0 |
| dfdiv | *Commercial Tool* | 1917 | 81.70 | 24.46 | 4074 | 2726 | 0 | 24 |
| | *SDC* | 1815 | 73.83 | 24.58 | 2539 | 1738 | 0 | 18 |
| | *Default* | 1823 | 80.78 | 22.57 | 2583 | 1786 | 0 | 18 |
| | *SDC* | 1815 | 73.83 | 24.58 | 2539 | 1738 | 0 | 18 |
| dfmul | *Commercial Tool* | 196 | 60.70 | 3.22 | 2433 | 1143 | 0 | 16 |
| | *Sequential* | 196 | 97.24 | 2.02 | 1452 | 855 | 0 | 10 |
| | *Default* | 113 | 77.95 | 1.45 | 1431 | 694 | 0 | 10 |
| | *SDC* | 105 | 73.41 | 1.43 | 1364 | 620 | 0 | 10 |
| dfsin | *Commercial Tool* | 48226 | 52.50 | 918.59 | 11491 | 6041 | 4 | 43 |
| | *Sequential* | 56625 | 70.23 | 806.28 | 11271 | 8572 | 0 | 41 |
| | *Default* | 46823 | 70.99 | 659.60 | 10417 | 6491 | 0 | 41 |
| | *SDC* | 44919 | 69.78 | 643.69 | 11375 | 5412 | 0 | 41 |
| gsm | *Commercial Tool* | 3397 | 66.40 | 51.16 | 4569 | 1391 | 9 | 40 |
| | *Sequential* | 2866 | 72.54 | 39.51 | 4652 | 2836 | 5 | 42 |
| | *Default* | 2796 | 68.16 | 41.02 | 4844 | 2865 | 3 | 29 |
| | *SDC* | 2328 | 71.96 | 32.35 | 4446 | 2710 | 3 | 30 |
| jpeg | *Commercial Tool* | 468011 | 69.00 | 6782.76 | 18347 | 9683 | 55 | 14 |
| | *Sequential* | 452181 | 71.81 | 6296.62 | 15928 | 9167 | 58 | 8 |
| | *Default* | 445967 | 72.76 | 6129.37 | 15710 | 8675 | 58 | 8 |
| | *SDC* | 428762 | 73.56 | 5828.59 | 15794 | 8653 | 58 | 8 |
| mips | *Commercial Tool* | 2489 | 77.60 | 32.07 | 1359 | 511 | 4 | 8 |
| | *Sequential* | 3098 | 73.15 | 42.35 | 1496 | 590 | 4 | 8 |
| | *Default* | 2867 | 74.82 | 38.32 | 1564 | 569 | 4 | 8 |
| | *SDC* | 2493 | 75.18 | 33.16 | 1524 | 539 | 4 | 8 |
| mpeg2 | *Commercial Tool* | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| | *Sequential* | 4227 | 89.21 | 47.38 | 1908 | 1272 | 2 | 0 |
| | *Default* | 4226 | 94.80 | 44.58 | 1952 | 1238 | 2 | 0 |
| | *SDC* | 4222 | 71.59 | 58.97 | 1885 | 1117 | 2 | 0 |
| sha | *Commercial Tool* | 111043 | 97.90 | 1134.23 | 2180 | 1356 | 20 | 0 |
| | *Sequential* | 114356 | 88.26 | 1295.65 | 2530 | 2239 | 12 | 0 |
| | *Default* | 113326 | 82.64 | 1371.24 | 2326 | 2041 | 12 | 0 |
| | *SDC* | 113322 | 76.01 | 1490.86 | 2384 | 2138 | 12 | 0 |

- *Final Size of Graph of Passes* i.e., the number of vertices and edges of the graph at the end of the execution of the design flow.
- *Dynamic Size of the Design Flow* i.e., the number of executed and skipped passes and the number of times the structure of the *Graph of Passes* is modified during the design flow.
- *Execution Times* of the updating of the *Graph of Passes*, of the High Level Synthesis Flow, and of the Logic Synthesis plus the Place-and-Route.

The number of vertices of the *Graph of Passes* for the *Sequential* flow and the *Default* flow is the same, while the number of edges is slightly different since the second also includes invalidation edges. The *SDC* flow adds roughly two more vertices for each function composing the benchmark, so the final size of the graphs for all the three flows is very similar. The two added passes are *SDC*

Table 6. Characteristics of the three design flows when targeting Stratix V board. *|V|*: the size of the *Graph of Passes* in terms of the number of vertices. *|E|*: the size of the *Graph of Passes* in terms of the number of edges. *Exec*: the overall number of executions of passes. *Skip*: the overall number of executions of passes that are skipped since not necessary. *Mods*: the overall number of times the *Graph of Passes* is modified. *PG time*: the execution time spent in updating the *Graph of Passes* and selecting the next pass to be executed. *HLS time*: the execution time of the design flow implemented in Bambu. *LS Time*: the execution time required by Logic Synthesis and Place-and-Route.

| | | Final Size of Graph of Passes | | Number of Passes | | | Execution Times (Seconds) | | |
|---|---|---|---|---|---|---|---|---|---|
| | Flow | \|V\| | \|E\| | Exec | Skip | Mods | PG | HLS | LS |
| adpcm_encode | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.00 | 5.12 | 578.15 |
| | *Default* | 151 | 636 | 383 | 49 | 26 | 0.02 | 6.49 | 553.03 |
| | *SDC* | 155 | 657 | 617 | 113 | 33 | 0.11 | 15.60 | 575.51 |
| aes_encrypt | *Sequential* | 566 | 2642 | 518 | 18 | 43 | 0.09 | 61.84 | 604.02 |
| | *Default* | 566 | 2673 | 1278 | 1145 | 71 | 0.26 | 63.39 | 608.36 |
| | *SDC* | 580 | 2766 | 2251 | 3083 | 93 | 0.66 | 116.16 | 607.95 |
| aes_decrypt | *Sequential* | 566 | 2642 | 518 | 18 | 43 | 0.11 | 43.73 | 614.68 |
| | *Default* | 566 | 2672 | 1228 | 998 | 69 | 0.30 | 46.40 | 615.51 |
| | *SDC* | 580 | 2765 | 2205 | 2996 | 92 | 0.66 | 99.18 | 618.94 |
| gsm | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.01 | 9.95 | 776.52 |
| | *Default* | 151 | 635 | 358 | 19 | 26 | 0.01 | 12.09 | 775.30 |
| | *SDC* | 155 | 656 | 492 | 32 | 29 | 0.07 | 26.50 | 737.50 |
| sha | *Sequential* | 317 | 1432 | 290 | 9 | 28 | 0.02 | 3.94 | 452.54 |
| | *Default* | 317 | 1447 | 636 | 273 | 41 | 0.08 | 6.16 | 472.78 |
| | *SDC* | 325 | 1497 | 1167 | 879 | 54 | 0.23 | 12.71 | 482.26 |
| blowfish | *Sequential* | 234 | 1030 | 214 | 6 | 23 | 0.03 | 81.69 | 625.74 |
| | *Default* | 234 | 1042 | 509 | 117 | 35 | 0.05 | 64.89 | 636.41 |
| | *SDC* | 240 | 1077 | 886 | 395 | 44 | 0.14 | 101.49 | 652.49 |
| dfadd | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.00 | 3.15 | 603.80 |
| | *Default* | 151 | 636 | 471 | 53 | 29 | 0.04 | 26.00 | 606.85 |
| | *SDC* | 155 | 657 | 995 | 116 | 45 | 0.07 | 84.62 | 650.94 |
| dfdiv | *Sequential* | 234 | 1025 | 213 | 6 | 25 | 0.02 | 3.42 | 613.86 |
| | *Default* | 234 | 1034 | 575 | 206 | 37 | 0.08 | 9.48 | 646.22 |
| | *SDC* | 240 | 1067 | 1043 | 472 | 49 | 0.20 | 21.83 | 662.93 |
| dfmul | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.00 | 2.29 | 497.50 |
| | *Default* | 151 | 636 | 472 | 53 | 29 | 0.00 | 7.37 | 503.54 |
| | *SDC* | 155 | 657 | 682 | 102 | 35 | 0.07 | 15.45 | 542.63 |
| dfsin | *Sequential* | 317 | 1425 | 289 | 9 | 30 | 0.01 | 11.71 | 1161.23 |
| | *Default* | 317 | 1440 | 857 | 416 | 50 | 0.08 | 49.72 | 1100.51 |
| | *SDC* | 325 | 1487 | 1706 | 1185 | 72 | 0.32 | 212.92 | 1078.27 |
| jpeg | *Sequential* | 566 | 2640 | 518 | 18 | 43 | 0.06 | 31.58 | 1268.09 |
| | *Default* | 566 | 2674 | 1609 | 1596 | 82 | 0.32 | 37.86 | 1249.43 |
| | *SDC* | 580 | 2771 | 2677 | 4078 | 107 | 0.96 | 55.89 | 1237.66 |
| mips | *Sequential* | 234 | 1030 | 214 | 6 | 23 | 0.01 | 3.22 | 479.02 |
| | *Default* | 234 | 1039 | 358 | 80 | 28 | 0.01 | 3.28 | 491.24 |
| | *SDC* | 240 | 1076 | 758 | 358 | 39 | 0.13 | 13.04 | 513.99 |
| motion | *Sequential* | 649 | 3064 | 594 | 21 | 48 | 0.08 | 4.45 | 418.25 |
| | *Default* | 649 | 3100 | 1382 | 1244 | 77 | 0.32 | 4.23 | 421.05 |
| | *SDC* | 665 | 3218 | 2525 | 4469 | 103 | 0.70 | 6.88 | 413.57 |
| satd | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.03 | 7.37 | 481.52 |
| | *Default* | 151 | 631 | 159 | 4 | 19 | 0.03 | 5.37 | 488.84 |
| | *SDC* | 155 | 651 | 244 | 14 | 21 | 0.02 | 13.59 | 495.21 |
| sobel | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.00 | 2.50 | 344.70 |
| | *Default* | 151 | 635 | 289 | 15 | 23 | 0.04 | 3.40 | 332.20 |
| | *SDC* | 155 | 656 | 479 | 46 | 28 | 0.02 | 3.22 | 340.80 |
| bellmanford | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.01 | 1.50 | 430.80 |
| | *Default* | 151 | 634 | 194 | 7 | 20 | 0.01 | 1.51 | 432.72 |
| | *SDC* | 155 | 655 | 359 | 24 | 24 | 0.03 | 2.42 | 443.35 |
| matrix | *Sequential* | 151 | 630 | 138 | 3 | 18 | 0.00 | 1.27 | 364.96 |
| | *Default* | 151 | 635 | 253 | 13 | 22 | 0.01 | 1.40 | 369.44 |
| | *SDC* | 155 | 656 | 405 | 45 | 26 | 0.03 | 1.88 | 378.16 |

Table 7. Characteristics obtained by LegUp and by the three design flows when targeting Stratix V.

| | Flow | Cycles | Freq. [MHz] | Wall-time [$\mu s$] | ALMs | 20K | DSPs |
|---|---|---|---|---|---|---|---|
| adpcom_encode | LegUp | 7883 | 245 | 32.12 | 2490 | 0 | 43 |
| | *Sequential* | 14319 | 256.67 | 55.79 | 1985 | 6 | 19 |
| | *Default* | 11882 | 270.64 | 43.90 | 2117 | 2 | 15 |
| | *SDC* | 11217 | 263.57 | 42.56 | 1989 | 2 | 18 |
| aes_encrypt | LegUp | 1564 | 395 | 3.96 | 4263 | 8 | 0 |
| | *Sequential* | 2525 | 442.28 | 5.71 | 4431 | 2 | 0 |
| | *Default* | 2476 | 446.63 | 5.54 | 4444 | 2 | 0 |
| | *SDC* | 2440 | 451.67 | 5.40 | 4411 | 2 | 0 |
| aes_decrypt | LegUp | 7367 | 313 | 23.56 | 4297 | 14 | 0 |
| | *Sequential* | 3349 | 364.70 | 9.18 | 4750 | 2 | 5 |
| | *Default* | 3282 | 376.93 | 8.71 | 4784 | 2 | 5 |
| | *SDC* | 3221 | 373.83 | 8.62 | 4757 | 2 | 5 |
| gsm | LegUp | 3966 | 273 | 14.52 | 4311 | 1 | 51 |
| | *Sequential* | 3416 | 281.93 | 12.12 | 3148 | 4 | 67 |
| | *Default* | 3322 | 271.74 | 12.22 | 3310 | 4 | 65 |
| | *SDC* | 2867 | 272.78 | 10.51 | 3224 | 4 | 45 |
| sha | LegUp | 168886 | 250 | 676.90 | 6398 | 26 | 0 |
| | *Sequential* | 225872 | 443.66 | 509.12 | 1606 | 11 | 0 |
| | *Default* | 224840 | 456.00 | 493.07 | 1549 | 11 | 0 |
| | *SDC* | 224836 | 454.13 | 495.09 | 1517 | 11 | 0 |
| blowfish | LegUp | 75010 | 468 | 160.22 | 1679 | 0 | 0 |
| | *Sequential* | 88790 | 408.83 | 217.18 | 3349 | 0 | 0 |
| | *Default* | 88653 | 397.61 | 222.96 | 3439 | 0 | 0 |
| | *SDC* | 88653 | 366.30 | 242.02 | 3442 | 0 | 0 |
| dfadd | LegUp | 650 | 252 | 2.58 | 2812 | 1 | 0 |
| | *Sequential* | 499 | 279.88 | 1.78 | 3444 | 0 | 0 |
| | *Default* | 363 | 282.49 | 1.29 | 3399 | 0 | 0 |
| | *SDC* | 349 | 290.28 | 1.20 | 3459 | 0 | 0 |
| dfdiv | LegUp | 2046 | 183 | 11.20 | 4679 | 4 | 42 |
| | *Sequential* | 1920 | 203.71 | 9.43 | 2749 | 0 | 12 |
| | *Default* | 1811 | 196.27 | 9.23 | 2936 | 0 | 12 |
| | *SDC* | 1812 | 196.46 | 9.22 | 2683 | 0 | 12 |
| dfmul | LegUp | 209 | 186 | 1.12 | 1464 | 1 | 28 |
| | *Sequential* | 174 | 227.32 | 0.77 | 1378 | 0 | 7 |
| | *Default* | 90 | 205.25 | 0.44 | 1353 | 0 | 7 |
| | *SDC* | 75 | 202.96 | 0.37 | 1385 | 0 | 7 |
| dfsin | LegUp | 57858 | 189 | 305.79 | 9099 | 3 | 72 |
| | *Sequential* | 56876 | 202.35 | 281.08 | 8913 | 0 | 28 |
| | *Default* | 48824 | 194.44 | 251.10 | 7974 | 0 | 28 |
| | *SDC* | 48128 | 191.39 | 251.47 | 7552 | 0 | 28 |
| jpeg | LegUp | 1128109 | 220 | 5126.14 | 16276 | 41 | 85 |
| | *Sequential* | 568969 | 236.52 | 2405.60 | 9119 | 57 | 28 |
| | *Default* | 544881 | 227.01 | 2400.20 | 9161 | 57 | 25 |
| | *SDC* | 513022 | 223.76 | 2292.70 | 8953 | 58 | 26 |
| mips | LegUp | 5989 | 487 | 12.30 | 1319 | 0 | 15 |
| | *Sequential* | 6551 | 488.76 | 13.40 | 1258 | 0 | 4 |
| | *Default* | 6420 | 482.16 | 13.32 | 1214 | 0 | 4 |
| | *SDC* | 5873 | 483.79 | 12.14 | 1141 | 0 | 4 |
| motion | LegUp | 66 | 338 | 0.20 | 6788 | 0 | 0 |
| | *Sequential* | 178 | 400.32 | 0.44 | 977 | 0 | 0 |
| | *Default* | 176 | 395.26 | 0.45 | 953 | 0 | 0 |
| | *SDC* | 168 | 411.18 | 0.41 | 881 | 0 | 0 |
| satd | LegUp | 46 | 288 | 0.16 | 2004 | 0 | 0 |
| | *Sequential* | 30 | 329.16 | 0.09 | 2492 | 0 | 0 |
| | *Default* | 30 | 328.41 | 0.09 | 2519 | 0 | 0 |
| | *SDC* | 30 | 329.92 | 0.09 | 2547 | 0 | 0 |
| sobel | LegUp | 7561317 | 336 | 22502.58 | 1241 | 0 | 36 |
| | *Sequential* | 17962025 | 436.30 | 41168.96 | 672 | 0 | 3 |
| | *Default* | 16647933 | 455.17 | 36575.51 | 653 | 0 | 3 |
| | *SDC* | 15607023 | 375.66 | 41545.90 | 595 | 0 | 2 |
| bellmanford | LegUp | 2444 | 332 | 7.37 | 493 | 0 | 0 |
| | *Sequential* | 4776 | 370.51 | 12.89 | 688 | 0 | 0 |
| | *Default* | 4775 | 376.79 | 12.67 | 666 | 0 | 0 |
| | *SDC* | 4482 | 386.55 | 11.59 | 644 | 0 | 0 |
| matrix | LegUp | 101442 | 401 | 253.00 | 225 | 0 | 2 |
| | *Sequential* | 133186 | 459.56 | 289.81 | 246 | 0 | 2 |
| | *Default* | 133154 | 348.07 | 382.55 | 246 | 0 | 2 |
| | *SDC* | 133154 | 374.81 | 355.25 | 253 | 0 | 2 |

*Scheduling* and *SDC Code Motion* which have been detailed in Section 4. On the contrary, the number of execution of passes in the three flows is very different: the *SDC* flow executes up to almost ten times the passes of the *Sequential* flow (8.94x in case of dfadd). Since the number of instantiated passes is almost the same, this growth is mainly caused by the re-execution of passes. The number of skipped passes in case of *Sequential* flow is almost 0 since the sequential flow is composed of a sequence of passes applied once. The cyclic dependencies added in the *Default* flow introduce the skipping of some executions of some passes. If the first pass of a chain of passes repeatedly executed does not change the intermediate representation, the following passes of the chain can be skipped. The number of skipped passes in case of *SDC* flow is instead much larger (up to 59.10% of the total number of passes in case of jpeg): in this case, indeed, a High Level Synthesis pass invalidates a large part of the Middle-End optimization passes. The invalidation is not applied only to the passes of the same function, but also to the passes of the functions which call it since inter-procedural analyses are applied by some Middle-End Optimization passes. Nevertheless, in most of the cases, the transformation performed on the called functions does not impact the inter-procedural analysis results so that all the passes which depend on their results have not to be re-executed.

Figure 2 better highlights how the *Graph of Passes* evolves during the execution of different types of design flows. Black lines describe the overall number of vertices in the graph at each iteration. At the beginning of the design flow, the *Graph of Passes* contains only the *Composed* passes describing the different sub-flows and the passes which work on the whole specification. Then, new passes are mainly added after that call graph has been computed and when the prerequisites of *Composed* passes are recomputed. Finally, a limited number of passes is also added during the rest of the design flow required by single optimization passes. On the contrary, there are significant differences in the evolution of the number of executed passes (i.e., the union of *Success*, *Unchanged*, and *Skipped*) described by the red lines. In the *Sequential* flow, roughly at each iteration a new pass is executed or skipped, so the number of executed passes linearly grows. In the *Default* flow, each time a pass invalidates some of the predecessors, the number of passes marked as executed is decremented. However, the oscillations in the number of executed passes are relatively limited. The largest number of invalidations occurs when memory allocation invalidates bitsize analysis and all its successors. Finally, large and multiple oscillations can be noticed analyzing the graph of the *SDC* flow. Each time the *SDC Scheduling* pass of a function is executed, it invalidates all the other High Level Synthesis passes and some *Middle-End* passes. Note that only a subset of the invalidated passes is re-executed while the other can be skipped. Indeed, the overall number of skipped passes is quite large (4469).

By analyzing the execution times reported in Table 4, it can be noticed how the proposed approach, despite its larger complexity with respect to other pass managers, introduces a very small overhead in the execution time of the High Level Synthesis (less than 1% in most of the cases). Note that since the measured execution times are the aggregation of very small quantities, the reported values of PG execution times are significantly subject to noise and approximations, justifying that the reported overhead for *Sequential* flow is larger than *SDC* in some cases. While the overall execution times of High Level Synthesis of the *Sequential* flow and of the *Default* flow are very similar, the execution times of the *SDC* flow are significantly larger. The increment is only partially caused by the repetition of passes: on the dfsin benchmark, for example, the *SDC* flow executes 90% more passes than the *Default* flow, but the execution time is increased by a 6.17 factor. The main cause of this increment is the SDC scheduling pass added in the SDC flow, which, including the solution of an integer linear programming formulation, can be longer than the rest of the High Level Synthesis flow.

Table 4 reports also the time of the Logic Synthesis and Place-and-Route when applied to the outcomes of the three design flows applied to the different benchmarks. It can be noticed how in most of the cases the execution time of the Logic Synthesis and Place-and-Route is two orders of magnitude larger than the corresponding High Level Synthesis execution time, so that slowing of the latter does not impact significantly the overall design time of a hardware accelerator.

Table 5 shows the results obtained with a commercial tool and each design flow applied to each benchmark. It is worth noting that even if the commercial tool contains a pass manager based on a static sequence of passes, this cannot be considered as a completely fair baseline since the set of analyses and optimizations implemented in the two flows can be different. Nevertheless, the results show how the dynamic graph of passes allows High Level Synthesis to improve results even if the quality of results of the static design flow is already good. Data about blowfish and mpeg2 obtained with commercial tool have not been reported since the co-simulation of the generated hardware accelerators failed.

In the CHStone suite, two classes of benchmarks can be identified. The first class consists of the benchmarks which are dominated by the control: adpcm, dfadd, dfdiv, dfmul, dfsin, gsm, and mips. The second class instead consists of the benchmarks which are dominated by the memory accesses: aes, blowfish, jpeg, mpeg2, and sha. The three analyzed design flows mainly differ in the optimization of the Control Flow Graph. Indeed, the benchmarks belonging to the first class are the ones which get more advantage by the more optimizing design flows. In the best case (dfadd) the gain with respect to *Sequential* flow is 54% in terms of performance and 27% in terms of area. On the contrary, the *Default* flow and the *SDC* flow do not introduce any significant benefit on the benchmarks belonging to the second class, both in terms of performance and area. Since the characteristics which distinguish the two classes of benchmarks can be easily identified analyzing their intermediate representation, it could be possible to apply the different design flows on the different benchmarks. The proposed infrastructure easily allows the implementation of such scenario: an analysis pass classifying applications can be implemented and then the prerequisites of the *Composing pass* representing the High Level Synthesis flow would change according to the outcome of this analysis pass. However, the classification of applications is never so strict: the jpeg benchmark, for example, while it is mainly dominated by memory accesses, also includes a portion dominated by the control which gains benefits from the *SDC* flow. For this reason, the usually adopted approach in the High Level Synthesis flow is to perform the most optimizing design flow, even if on the particular application the benefits with respect to faster synthesis flows will not be so significant.

Table 6 reports the characteristics of the design flows in the second scenario. Even in this case, the size of the final graph of the three flows is not significantly different since the three flows share most of the passes. The largest difference is for the *SDC* flow of motion benchmark whose *Graph of Passes* adds 16 more passes than *Default* flow, which is still less than 3% of the overall size. On the contrary, the number of passes executed and skipped is significantly different since the invalidation causes the re-execution of several passes, in particular in the SDC flows. For example, in the synthesis of the benchmark with the largest number of functions (motion), the number of executed passes goes from 594 (*Sequential* flow) to 2525 (*SDC*) and the number of skipped passes goes from 21 (*Sequential*) to 4469 (*SDC*). The execution time of the PG is comparable to the results presented in Table 4 since the complexity of the updated *Pass of Graphs* is similar: in all the experiments is smaller than 1 second. Even in this scenario, SDC passes introduce a significant difference in the overall High Level Synthesis time. Finally, Logic Synthesis and Place-and-Route time is larger, but this depends on the usage of a different tool (Quartus vs. Vivado).

Table 7 presents the solutions obtained by the three design flows in terms of area and time. As a term of comparison, the results achieved with LegUp and presented in [18] have been included. It is worth noting that a direct comparison among the design flow engine proposed in this paper and the pass manager exploited in LegUp is not possible since the passes included in the two tools are different. Moreover, the solutions generated by Bambu are not necessarily the best ones in terms of Wall-time since changing the target frequency could produce better solutions. The results show a significant benefit both in terms of area and time in exploiting more complex design flows. Even in this scenario, the benchmarks whose implementations are much improved are the ones which are mainly characterized by control (e.g., adpcm, dfadd, dfmul). For example, the usage of the *SDC* more than doubles the performance of the dfmul with respect to the *Sequential* flow. Moreover, the target frequencies are significantly larger than the target used in the first scenario. They range from 183MHz to 487MHz while in the first scenario it 66MHz. Smaller clock periods reduce the possibility of chaining of operations in the same clock cycle resulting in fewer opportunities of profitable code motions. For this reason, the overall gain in terms of performance provided by the SDC flow in terms of clock cycles is smaller than in the first scenario but still significant.

## 6 CONCLUSIONS

The High Level Synthesis flow shares some characteristics of traditional compilers but, differently from them, can require complex sequences of optimizations which have to be customized for the single functions. The pass managers implemented in compilers have been designed to efficiently execute static sequences of passes, so they are not suitable to manage this type of scenarios. In this paper, a design flow engine designed for the execution of complex synthesis flows has been presented. This engine is based on a cyclic graph representing the relationships among passes and it supports dynamic addition of passes, cyclic dependencies, composed passes, selective invalidation and skipping of passes. The experimental results show how complex design flows can increase the quality of the generated designs without significantly impacting the execution time of the overall design flow. Future works will be about the possibility of integrating at runtime new passes in the infrastructure as external modules and about the parallelization of the execution of the passes of a design flow.

## REFERENCES

[1] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2, Article 21 (June 2016), 25 pages. https://doi.org/10.1145/2928270

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J.H. Anderson. 2013. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 24 (2013), 27 pages. https://doi.org/10.1145/2514740

[3] Deming Chen, Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. 2005. xpilot: A platform-based behavioral synthesis system. *SRC TechCon* 5 (2005).

[4] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE TCAD* 30, 4 (April 2011), 473–491.

[5] Jason Cong, Bin Liu, Raghu Prabhakar, and Peng Zhang. 2012. A Study on the Impact of Compiler Optimizations on High-Level Synthesis. In *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers.* 143–157. https://doi.org/10.1007/978-3-642-37658-0_10

[6] Jason Cong and Zhiru Zhang. 2006. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *2006 43rd ACM/IEEE Design Automation Conference.* ACM, New York, NY, USA, 433–438. https://doi.org/10.1145/1146909.1147025

[7] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2005. ACME: Adaptive Compilation Made Efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '05).* ACM, New York, NY, USA, 69–77. https://doi.org/10.1145/1065910.1065921

[8] Pietro Fezzardi, Michele Castellana, and Fabrizio Ferrandi. 2015. Trace-based automated logical debugging for high-level synthesis generated circuits. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015.* 251–258. https://doi.org/10.1109/ICCD.2015.7357111

[9] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla,

John Thomson, Christopher K. I. Williams, and Michael F. P. O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming* 39, 3 (2011), 296–327. https://doi.org/10.1007/s10766-010-0161-2

[10] GNU GCC. 2015. Plugin API. https://gcc.gnu.org/wiki/plugins.

[11] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *JIP* 17 (2009), 242–254.

[12] Scott Hauck and Andre DeHon. 2007. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. 2015. The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware. *ACM Trans. Reconfigurable Technol. Syst.* 8, 3, Article 14 (May 2015), 26 pages. https://doi.org/10.1145/2629547

[14] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. 2005. Fast and Efficient Searches for Effective Optimization-phase Sequences. *ACM Trans. Archit. Code Optim.* 2, 2 (June 2005), 165–198. https://doi.org/10.1145/1071604.1071607

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE Computer Society, Washington, DC, USA, 75–.

[16] Marco Lattuada and Fabrizio Ferrandi. 2015. Code Transformations Based on Speculative SDC Scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '15).* IEEE Press, Piscataway, NJ, USA, 71–77.

[17] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 7, 2, Article 6 (July 2014), 30 pages. https://doi.org/10.1145/2617593

[18] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (Oct 2016), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673

[19] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. 2016. A Graph-based Iterative Compiler Pass Selection and Phase Ordering Approach. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES 2016).* ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2907950.2907959

[20] C. Pilato and F. Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications (FPL '13).* 1–4. https://doi.org/10.1109/FPL.2013.6645550

[21] Richard M. Stallman and the GCC Developer Community. 2015. GNU GCC Speedup Areas. https://gcc.gnu.org/wiki/Speedup_areas.

[22] Richard M. Stallman and the GCC Developer Community. 2018. GNU Compiler Collection Internals. https://gcc.gnu.org/onlinedocs/gccint.pdf.

[23] J. Villarreal, A. Park, W. Najjar, and R. Halstead. 2010. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on.* 127–134. https://doi.org/10.1109/FCCM.2010.28

[24] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. 1994. *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler.* Technical Report. Computer Systems Laboratory - Departments of Electrical Engineering and Computer Science - Stanford University, Stanford, CA, USA.

[25] Xilinx. 2017. Vivado Design Suite. http://www.xilinx.com.