

FACETs: Fast and Efficient Compilation for Control–Dataflow Mapping on CGRAs

Yuxuan Wang^{1*}, Rubén Rodríguez Álvarez¹, Cristian Tirelli²,
Rodrigo Otoni³, Giovanni Ansaloni¹, Laura Pozzi², and David Atienza¹

¹ EPFL, Lausanne, Switzerland

² Università della Svizzera italiana, Lugano, Switzerland

³ University of Groningen, Groningen, Netherlands

In the proceedings of ARC’26: https://doi.org/10.1007/978-3-032-29365-7_5

Abstract. Reconfigurable computing bridges the gap between the flexibility of general-purpose processors and the efficiency of application-specific hardware. Among reconfigurable architectures, Coarse-Grained Reconfigurable Arrays (CGRAs) are particularly promising because they offer high efficiency and low configuration overhead, while maintaining programmability at the operation level. However, mapping high-level applications onto CGRAs remains challenging, as most state-of-the-art approaches either have a limited compilation scope or incur a very high compilation time. To address this challenge, we propose a compilation framework that supports generic control–dataflow graphs by scheduling dataflow graphs under explicit control-flow constraints. The framework adopts a flexible approach that comprises a fast heuristic mapping strategy for non-critical code regions, while enabling aggressive optimizations for performance-critical regions. Experimental results show up to 10× faster compilation for applications with complex control–dataflow graph structures, while achieving comparable or better runtime performance relative to existing approaches.

1 Introduction

Modern computing workloads require hardware platforms that deliver high performance, energy efficiency, and flexibility. Coarse-Grained Reconfigurable Arrays (CGRAs) address these requirements by combining efficient parallel execution with architectural reconfigurability [1]. Composed of a grid of reconfigurable processing elements (PEs), CGRAs execute time-multiplexed operations to achieve high performance and energy-efficient data propagation [2].

CGRAs compilers are tasked with mapping applications in PEs at given time slots, taking into account both spatial (e.g. connectivity) and temporal (e.g. def-use relationships) constraints. To this end, applications are usually translated from high level languages such as C to Intermediate Representations (IRs) using standard compilation chains. IR statements are then mapped on hardware resources [3]. Goal of the CGRA compilation process is to efficiently deploy

* Corresponding author: yuxuan.wang@epfl.ch

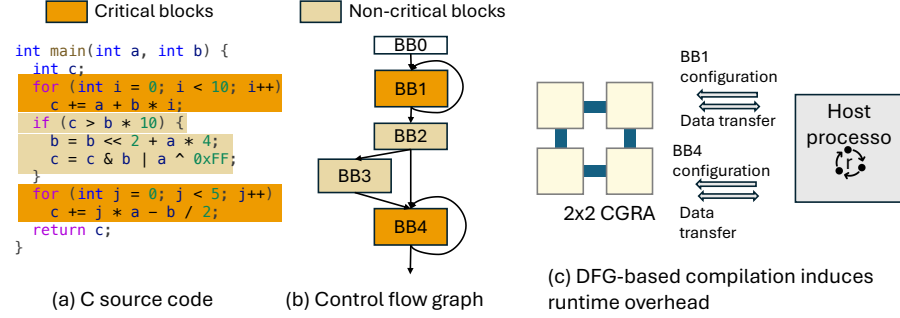


Fig. 1: Deployment of applications onto CGRA: limited mapping ranges introduce high reconfiguration and data transfer overhead at runtime.

operations on the available PEs, effectively exploiting parallelism. A prevalent compilation strategy is modulo scheduling (MS), which targets control-free loops represented as dataflow graphs (DFGs) [4]. This restriction limits the range of applications that can be mapped onto CGRA hardware. Consequently, application developers must either manually or automatically isolate compilable regions [5], and additional runtime overhead is introduced due to frequent reconfiguration. Figure 1 illustrates this limitation with an application composed of five basic blocks (BBs). As shown in Fig. 1(b), only BB1 and BB4 are mapped onto the CGRA, while the remaining blocks execute on the host processor. Hence, DFG-based approaches require the blocks to be compiled and executed independently. As depicted in Fig. 1(c), instructions configuration and data transfer are required before/after the CGRA execution of each BB. Even when BB4 consumes the data produced by BB1, the data must be transferred back to the host and reloaded, leading to redundant reconfiguration and unnecessary runtime overhead.

Control dataflow graph (CDFG) compilation expands the mapping scope to support general applications by enabling multiple basic blocks (e.g., BB0–BB4 in Figure 1) to be mapped onto the CGRA. Marionette [6] adopts this approach by introducing dedicated reconfiguration hardware to support basic block switching, which reduces—but does not eliminate—reconfiguration overhead. In contrast, CDFG-aware compilation eliminates runtime reconfiguration by switching basic blocks via the program counter [7]. However, this approach results in a very high compilation time as a result of the high mapping complexity. Moreover, a single high-complexity basic block can prevent the entire CDFG from being successfully compiled. For example, if BB3 in Figure 1 is too complex for the compiler to map, the entire application cannot be mapped to the CGRA.

Our approach is driven by the insight that critical blocks in a CDFG constitute only a subset of all basic blocks, and that this subset has a large impact on overall performance. As shown in Figure 1, the blocks highlighted in orange correspond to critical blocks that dominate runtime due to frequent execution and therefore determine the achievable performance of the application [8]. For these blocks, longer compilation time is justified to enable performance optimization

using techniques such as modulo scheduling. In contrast, low-impact blocks (e.g., BB2 and BB3) are executed infrequently—often only once or not at all—and have negligible influence on overall performance. For such blocks, it is acceptable to tolerate suboptimal mappings that incur a few additional clock cycles, as their impact on end-to-end performance is negligible.

Building on these observations, we propose FACETs - a CDFG mapping approach that maps each basic block while preserving correct data propagation across blocks. Inter-block dependencies are captured at basic block boundaries, enabling efficient CGRA execution as a single accelerated CDFG. FACETs is based on a heuristic approach that reduces compilation complexity and expands the compilation scope to CDFG, while enabling the integration of high-performance, but time-consuming, compilers for critical BBs only. Moreover, it effectively reduces the runtime required for data transfers between the host processor and the CGRA and for reconfiguration.

2 Related Works

2.1 DFG mappers

Existing DFG mapping techniques for CGRAs primarily differ in how they formulate and solve the spatial-temporal assignment of operations to PEs under data-dependency and routing constraints. Exact-based approaches can be broadly classified into Integer Linear Programming (ILP) formulations [9, 10] and Satisfiability Modulo Theories (SMT)-based methods [11–13]. These methods determine spatial and temporal mappings by solving explicitly constructed optimization models. However, their computational complexity grows exponentially with increasing problem size [14]. To alleviate this limitation, another class of mappers adopts heuristic placement policies guided by cost functions to determine the placement of DFG nodes on CGRA processing elements [15]. In this category, E2EMap [16] leverages reinforcement learning to learn placement policies for DFG mapping, where neural networks are trained through reward-driven optimization to guide placement decisions. Although the generated solutions are effective, training such models is challenging, and the mapping time typically ranges from 10^3 to 10^4 seconds [17, 18]. Moreover, DFG-only mapping approaches face inherent limitations in supporting broader program scopes. These issues stem from the restricted mapping scope of DFG mappers, which typically operate on a single loop body: data initialization and write-back operations often lie outside the loop DFG and therefore cannot be mapped automatically.

2.2 Control dataflow compilation

To address these limitations, compilation must be extended beyond isolated DFGs to the CDFG level. HDCC [19] addresses this by identifying DFGs suitable for CGRA deployment and analyzing their live-in and live-out values to insert the required load and store operations. This automates memory accesses and accelerates only the selected kernels, while leaving a substantial portion of the application to execute on the host processor.

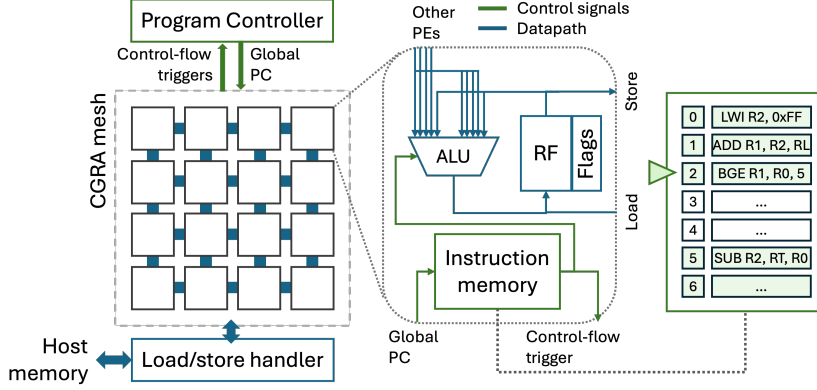


Fig. 2: Architectural overview of the targeted instruction-based CGRA. The datapath is represented in blue, while control-flow signals are depicted in green.

To overcome this limitation, several works propose specialized architectures and compilation techniques targeting common control patterns, such as divergent branches and nested loops. Cao et al. [20] propose a Speculative Iteration Execution (SIE) CGRA architecture that decouples path selection from condition evaluation. In this design, the SIE controller speculatively issues and executes loop iterations, enabling support for divergent branches within loops. However, this approach requires rollback mechanisms when predication errors occur, constraining its applicability and efficiency. The Adora compiler [21] employs loop transformations to enable efficient dataflow-level execution and optimizes data communication at the task level, with a particular focus on improving the execution of nested loop structures.

These approaches typically treat common control-flow patterns, such as if-then-else constructs and nested loops, as independent problems, leaving more general cases—such as dynamic-boundary loops or multiple branches—unaddressed. Conversely, Marionette [6] relies on reconfiguration-based techniques that can adapt the CGRA configuration at runtime according to conditions, but they incur considerable hardware and configuration overhead. Finally, static approaches that compile the entire CDFG with general control flows are often based on ILP models, which result in long compilation time [22].

3 Target CGRA Architecture

We assume as compilation target a prototypical CGRA consisting of a 2D mesh of interconnected PEs. Each PE contains a register file (RF) with data and zero/negative/overflow fields, an ALU, and a private instruction memory. To support CDFG execution, a program counter (PC) directs the control flow, as illustrated in Figure 2. Supported operations include arithmetic and logic operations, flag-based select operations, conditional branches, jumps and load/store accesses to host memory.

At each instruction step, each PE issues one operation from its private instruction memory as dictated by a global program counter. All PEs simultaneously process their current instruction. ALU operations may consume operands either from the local RF or from other PEs, and the result is written back to the RF. A memory controller interfaces PEs to memory for load and store operations. Once all PEs complete their current operation, the program controller updates the global PC to the next instruction. Each PE can overwrite the PC by issuing a control operation (i.e., branch, jump, and exit). For example, in PC=2 in Figure 2, if the value in $R1 \geq R0$, the next value for the PC is set to PC=5. Such mechanism enables the deployment of applications with generic control flows, including if-clauses and loop nests.

4 Methodology

FACETs manages data placement and propagation at the CDFG level, enabling seamless BB transitions via the PC and eliminating reconfiguration overhead during BB switches. Specifically, Section 4.1 presents our approach to manage inter-basic-block data propagation in the CDFG. Section 4.2 details the scheduling strategy used to efficiently map each basic block, and Section 4.3 describes the data propagation mechanism that updates shared constraints after the DFG of a basic block has been scheduled.

4.1 Overview: Control flow management

The CDFG captures inter-block data propagation subject to the underlying CDFG structure, which consists of multiple basic blocks connected by control operations that define block transitions. Each block contains a DFG that is executed when control reaches the block—for example, when a predecessor executes a branch operation targeting it. Figure 3 illustrates an example of mapping a CDFG onto CGRA hardware, with the CDFG shown in Figure 3a. The bold grey lines indicate the CDFG transitions of an outer loop with a divergent branch ($BB0 \rightarrow BB1 \rightarrow BB3$ or $BB0 \rightarrow BB2 \rightarrow BB3$). In this structure, one path forms a self-loop within $BB2$, while the thin black lines represent the corresponding data dependencies. Within this execution flow, $BB0$ and $BB2$ evaluate the control conditions that determine which successor block is triggered.

To manage control flow while coordinating the DFG execution within each basic block, we define two graph states for every basic block to represent the spatial mapping of operations onto the CGRA hardware:

- **Entry graph** ($\mathcal{G}^E$): the spatial mapping graph when PC enters a block.
- **Exit graph** ($\mathcal{G}^X$): the spatial mapping graph after the DFG of the basic block has completed execution.

Figure 3b and Figure 3c present the entry and exit graphs of each basic block, integrating both control and data-flow operations to support CDFG mapping. The objective of this spatial mapping is to ensure that values produced or propagated in a basic block are available at the end of its execution such that they can

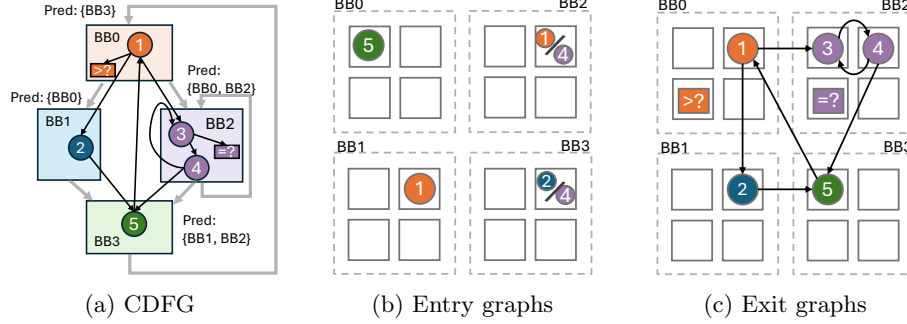


Fig. 3: Mapping solutions for a nested loop with divergent branches with livein/liveout mapped in the BB boundaries: the control operation determines the next BB to be executed. (Op $\textcircled{i}$ /Op $\textcircled{j}$) denotes either $\textcircled{i}$ or $\textcircled{j}$ is used, depending on the control flow.)

be correctly consumed by operations in successor DFGs. For example, the consumer operations in BB1 and BB2 can access the producer value (operation $\textcircled{1}$) defined in BB0, regardless of whether control transfers, as shown in their entry graphs. Operation $\textcircled{5}$ receives its input from operation $\textcircled{2}$ when BB1 is taken, or from operation $\textcircled{4}$ when BB2 is taken; in both cases, the value is routed from the right neighboring processing element, as shown in the exit graphs.

To guarantee the correct execution of a DFG, its entry graph must provide all live-in values required for computation. Accordingly, control-flow management across basic blocks is defined by the following constraint:

$$\mathcal{G}_{B_n}^E \subseteq \mathcal{G}_{B_m}^X, \quad B_m \in \text{Pred}(B_n) \quad (1)$$

The entry graph of a block is a subgraph of its predecessors' exit graphs, ensuring that live-in values are available to perform the block's computation. For example, BB3 has two predecessors BB1 and BB2. Therefore, the entry graphs of BB3 should be the subset of both the exit graphs of BB1 and BB2.

4.2 Data flow mapping

Data flow graph mapping incorporates entry and exit graph prerequisites that encode CDFG-level constraints from other basic blocks, producing temporal-spatial solutions that conform to these requirements. Figure 4 shows the heuristic process of mapping one DFG to the hardware, considering the CDFG constraints.

Temporal Scheduling Candidates An operation is eligible for scheduling if and only if all of its producer values are available in the CGRA registers. Under the as-soon-as-possible (ASAP) policy, all schedulable operations are first

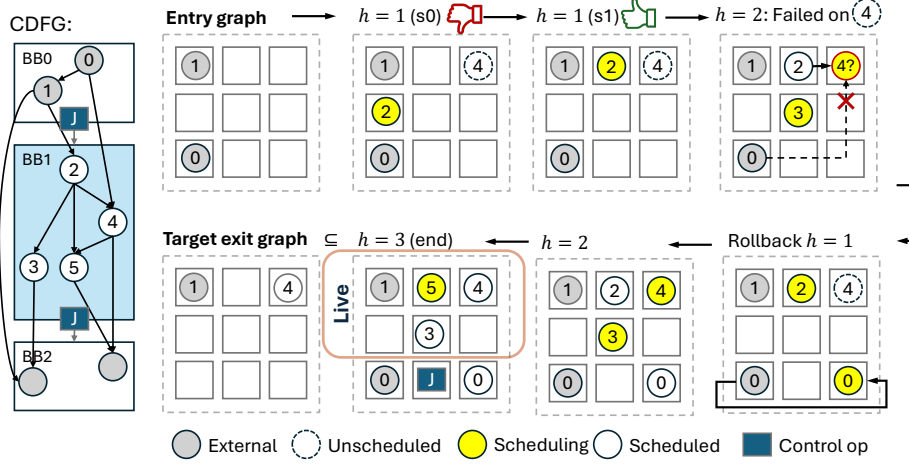


Fig. 4: Heuristic mapping process of BB1, following the prerequisites of its entry graph and exit graph.

retrieved, thereby allowing greater flexibility in the subsequent mapping decisions. The yellow operations in Figure 4 show the set of temporal scheduling candidates in each height, which are denoted as $\mathcal{I}_C$ and defined as follows:

$$\mathcal{I}_C = \{I_k | \text{Prod}(I_k) \subseteq V_I \cup \mathcal{I}_D, \}, \quad (2)$$

where $\text{Prod}(I_k)$ denotes the producer operations of the candidate operation I_k , $\mathcal{I}_D$ is the set of operations already scheduled within the current DFG, and V_I represents the set of live-in values that are not produced internally within the basic block. More generally, an operation becomes schedulable once all of its producers are either live-in values or operations that have already been scheduled in the same DFG. The availability of live-in values is ensured by Equation 1, which guarantees that they are prepared in the entry graph.

Control operations (e.g., branch or jump instructions) are required to be scheduled at the end of each basic block, following the conventional compiler design principle. This constraint arises because control operations determine the next PC value and thus define the boundary of a basic block. Scheduling control operations earlier would prematurely redirect control flow, potentially preventing the execution of remaining data operations in the block and violating the single-entry–single-exit semantics of a basic block. Moreover, in CGRA architectures, the effects of control operations must be resolved only after all data computations in the block have completed, ensuring that live-out values are fully produced before control is transferred to successor blocks.

Spatial Placement Space The spatial placement space of an operation is determined by both data dependencies and the architectural constraints of the

CGRA. An operation must be placed on a PE that can receive its input operands from the PEs hosting its producer operations and forward its output values to the PEs hosting its consumer operations, subject to the availability of routing resources. In addition, the selected PE must provide sufficient register capacity to store newly generated live values without overwriting existing ones. Formally, for an operation I , its spatial placement space $\mathcal{S}(I)$ is defined as

$$\mathcal{S}(I) = \mathcal{A}_{\text{Prod}}(I) \cap \mathcal{A}_{\text{Cons}}(I) \cap \mathcal{A}_{\text{Reg}}, \quad (3)$$

where $\mathcal{A}_{\text{Prod}}(I)$ and $\mathcal{A}_{\text{Cons}}(I)$ denote the sets of PEs that can communicate with the scheduled producer and consumer operations of I , respectively, and $\mathcal{A}_{\text{Reg}}$ denotes the set of PEs with sufficient register resources to accommodate additional live values.

Both producer- and consumer-side constraints are considered because the BB scheduling does not strictly follow dominance order, which traditionally requires that a block in the control-flow graph be scheduled only after all of its dominating predecessors have been placed. Therefore, a producer operation from other BB may not yet be assigned to a physical PE and does not impose immediate spatial restrictions on its consumers. Conversely, once a consumer operation has been spatially scheduled, its placement introduces concrete communication constraints that must be satisfied by its producers. For example, as shown in Figure 3, when the DFG of BB2 is scheduled first and the placement of operation ① in BB0 remains undecided, the mapping of operation ③ retains greater spatial flexibility. However, when operation ① is scheduled at a later stage, it must be placed in a manner that satisfies the constraints established by operation ③.

This scheduling strategy enables BB scheduling without strictly adhering to dominance order. By deferring the spatial binding of values originating from unscheduled basic blocks, the approach preserves greater placement flexibility for performance-critical regions such as loops. The algorithm for maintaining Equation 1 after each basic block is scheduled is described in Section 4.3.

Height-Based Scheduling To enable CDFG mapping, DFG scheduling must explicitly account for inter-BB data dependencies. This is achieved by enforcing compatibility between the entry and exit graphs of the basic block being scheduled, as shown in Figure 4. Scheduling starts from the entry graph, which specifies live-in value placements, and aims to produce an exit graph satisfying successor-block requirements, with the target exit graph constrained to be a subgraph of the final exit graph after scheduling ($\mathcal{G}_{\text{target}}^X \subseteq \mathcal{G}^X$).

Algorithm 1 describes the DFG scheduling procedure under the constraints imposed by the entry and exit graphs. The algorithm initializes scheduling from the entry graph and proceeds in a layer-by-layer manner following data dependencies. At each scheduling height, it identifies the schedulable operation set $\mathcal{I}_C$ and randomly selects a spatial mapping from the placement scheduling space $\mathcal{S}$. To identify an optimal placement, the algorithm performs N_{SA} iterations, perturbing the placement within the search space to minimize the cost. In each

Algorithm 1: Priority-Guided Spatial-Temporal Scheduling

Input: Entry graph $\mathcal{G}^E$, Exit graph $\mathcal{G}^X$, Full operation sets I_{full} ,
Exploration steps N_{SA}

- 1 scheduled operation sets $I_S \leftarrow \emptyset$;
- 2 Unscheduled operation sets $I_U \leftarrow I_{full}$;
- 3 $h \leftarrow 1$; $\mathcal{G}^{before} \leftarrow \mathcal{G}^E$;
- 4 **while** I_U not empty **do**
- 5 Get schedulable operations at time h : $\mathcal{I}_C$;
- 6 Randomly place operations under spatial constraints $\mathcal{S}(I)$ to generate
current placement graph $\mathcal{G}_{after}$ and record initial cost C_{best} ;
- 7 **for** $iter = 1$ to N_{SA} **do**
- 8 Compute $\mathcal{S}(I)$ based on $\mathcal{G}^{before}$ and $\mathcal{G}^X$;
- 9 Randomly perturb placement within search space to get $\mathcal{G}^{cur}$;
- 10 Evaluate new cost C_{cur} ;
- 11 **if** $\text{ACCEPT}(C_{best}, C_{cur})$ **then**
- 12 $C_{best} \leftarrow C_{cur}$; $\mathcal{G}_{after} \leftarrow \mathcal{G}^{cur}$;
- 13 **if** convergence detected **then**
- 14 **break**;
- 15 $\mathcal{T} \leftarrow \{I \in \mathcal{I}_C \mid I \notin \text{scheduledOps} \wedge \text{latest schedule}(I) \leq h\}$;
- 16 **if** $\mathcal{T} \neq \emptyset$ **then**
- 17 Apply graph transformations;
- 18 Roll back schedule and graph to height h_{rb} ;
- 19 **continue**;
- 20 Commit scheduling results at time h ;
- 21 $I_D \leftarrow \mathcal{I}_C - \mathcal{T}$; $I_U \leftarrow I_U - I_D$
- 22 $\mathcal{G}^{before} \leftarrow \mathcal{G}_{after}$; $h \leftarrow h + 1$;
- 23 **return** $\mathcal{S}$ and $\{\mathcal{G}^{after}\}$

iteration, the resulting layout is evaluated using the following cost function:

$$\sum_{I_U} 2^{-priority} \log_2(1 + d) \quad (4)$$

$$priority = \frac{\max(h_0, h) + h_1}{2} \quad (5)$$

The scheduling priority reflects the urgency of scheduling an operation at height h , where h_0 and h_1 represent its earliest and latest possible schedule time, respectively. I_U denotes the set of unscheduled operations, and d is the distance between an unscheduled operation and its producers. This cost function evaluates the impact of the current layout on the remaining operations, accounting for both the availability of suitable PEs for direct mapping and the additional routing required when placement is constrained. Figure 4 illustrates this concept: at $h = 1$, operation ② is successfully mapped in both scenarios s1 and s2. However, in situation s1, operation ④ is positioned closer to its producer than in s0, yielding a lower cost and a more favorable layout, which is thus accepted.

If the scheduling space for an operation is empty, i.e., $S(I) = \emptyset$, the operation cannot be scheduled at the current height h . If $h < h_1$, scheduling can be deferred and potentially resolved in later layers. For example, if two operations must be executed on the same PE, the conflict can be resolved by executing one operation in the current layer and the other in a subsequent layer. However, if the operation has reached its latest allowable schedule time (i.e., it lies on the critical path), deferring is no longer feasible. As illustrated in Figure 4, failing to map operation ④ would block the scheduling of all subsequent operations, since it lies on the critical path. To resolve this, additional routing is performed. When scheduling the routing operation, the height is rolled back to $\max(h_p + 1, h - l_R)$, where h_p is the scheduling height of the producer and l_R is the routing length. This approach minimizes the rollback required for rescheduling while allowing routing operations to execute in parallel with other operations. As a result, the final temporal schedule is not affected, provided that the routing operations do not extend the critical path.

The placement is performed in a layer-by-layer manner until all operations are scheduled. The exit graph captures all live-out values required for the placement of successor blocks, including both the prerequisites specified by the target exit graph and the new values produced during scheduling. These values are then used to update the entry and exit graphs of all basic blocks to maintain CDFG consistency, as discussed in Section 4.3.

4.3 Update the entry and exit graphs

The scheduling of basic blocks derives their constraints from the corresponding entry and exit graphs, preserve consistency across the CDFG, i.e. if a basic block (B) produces an exit graph ($\mathcal{G}_B^X$), for each successor block ($S(B)$) the live-in values in its entry graph must conform to the operations mapped in $\mathcal{G}_B^X$. If a value is propagated through the successor block—i.e., it remains live in the successor’s exit graph—it must also live in its exit graph.

$$\mathcal{G}_{S(B)}^E \leftarrow \text{Live}(\mathcal{G}_B^X), \mathcal{G}_{S(B)}^X \leftarrow \text{Live}(\mathcal{G}_B^X) \quad (6)$$

Here, *Live* notates the live value sets that should be kept if it is used by its successors. This propagation process is applied recursively to successor blocks until no successors remain or a block has already been visited.

Similarly, if an operation assumes that a live-in value is sourced from a specific spatial PE, this requirement is propagated backward to its predecessor blocks to ensure that they produce consistent results. The exit graph of each predecessor is updated accordingly. If the required value is not produced within a given predecessor block, the update is further propagated through its predecessors. The backward propagation proceeds recursively until the block that produces the value is reached:

$$\mathcal{G}_{P(B)}^X \leftarrow \text{Live}(\mathcal{G}_B^E), \mathcal{G}_{P(B)}^E \leftarrow \text{Live}(\mathcal{G}_B^E) \quad (7)$$

CDFG consistency is maintained through updates to the entry and exit graphs, where unscheduled blocks only need to satisfy the prerequisites imposed

by scheduled blocks. In Figure 4, the yellow block highlights the live sets at the end of scheduling ($h = 3$), which are used to update the entry graph of its successor, BB2. Values that are no longer live and unused, such as Operation ⑩, do not affect subsequent scheduling.

This separation enables independent scheduling of different blocks: critical blocks can devote greater compilation effort to exploit optimizations. For example, modulo scheduling can be used to improve runtime performance without initially considering inter-basic-block constraints. Once these blocks produce their scheduling solutions, non-critical blocks can then be mapped using height-based scheduling while adhering to the generated constraints, thereby quickly obtaining feasible solutions. As a result, the overall compilation time is reduced, allowing the approach to scale to large CDFGs. Moreover, it helps avoid unnecessary communication overhead between the CGRA and the host processor.

5 Experiments

5.1 Experimental setup

Benchmarks: We evaluate our framework using a set of applications that exhibit a variety of control-flow patterns, selected from the MiBench suite [23] and Rodinia [24]. We select the benchmarks with divergent control flow patterns including multi-level nested and sequential loops, dynamic loop boundaries, runtime selection, and their combinations. The characteristics of the selected applications are summarized in Table 1, including the number of basic blocks, and the number of operations.

Table 1: Benchmarks.

Benchmarks	# BBs	# Ops	CFG attributes
bicg	7	61	2-level nested loops
2mm	17	107	Sequential nested loops
sha	5	41	Sequential loops
gsm	3	32	Loop with selection
hotspot3D	9	84	3-level nested loops with selection
isqrt	3	27	DFG loop with data initialization
lud	14	77	Sequential 3-level nested loops with dynamic boundaries

Baseline: We compare our methodology against Compigra [22]. Compigra is built on an ILP-based scheduling framework that consider for inter BB data propagation at compile time. For MS of loop DFGs, Compigra integrates SATMapIt [12], whereas our approach adopts MonoMap [25], which to the best of our knowledge is the fastest open-source MS tool. For assessing the runtime of scheduled application, we further compare with Marionette [6], which, as opposed to the

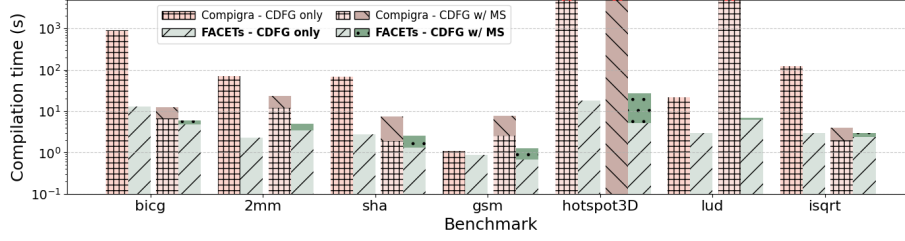


Fig. 5: Compilation time across benchmarks. For each benchmark, the four bars represent Compigra CDFG mapping, our CDFG mapping, Compigra CDFG mapping with MS, and our CDFG mapping with MS.

simple PC-based strategy of FACETs (Section 3) relies on specialized hardware support for control-flow management.

Testbed: Our experimental evaluation is conducted on a 4×4 CGRA implementation derived from OpenEdgeCGRA [26], an open-source, silicon-validated CGRA design. The PEs are arranged in a 2D mesh and connected using nearest-neighbor links with wrap-around connectivity. In the considered implementation, each PE has read-only access to the full RF of its neighboring PEs. Furthermore, PEs operate in lockstep under a shared global program counter, and all instructions complete in a single clock cycle. We use 6-entry register files for each PE and an instruction memory capacity of 128 instructions, which suffice for all considered benchmarks.

5.2 Compilation time

Figure 5 compares the compilation latency of our approach against Compigra under two settings: (1) standalone CDFG compilation with height-based scheduling introduced in Section 4.2, and (2) integrated compilation with MS for critical blocks. In both settings, our method consistently outperforms Compigra’s ILP-based mapping, typically completing in under 10 seconds, while DFG solvers such as Morpher [15] and E2EMap [16] require over 100 seconds for the DFG loop alone for **bicg**. While incorporating DFG optimization introduces a slight overhead, the impact is modest. With MS enabled, the compilation time for the remaining blocks is reduced, as the computation-intensive blocks are already mapped, leaving fewer blocks to schedule. For CDFG-only compilation with height-based scheduling, our approach is faster by several orders of magnitude (as shown by the logarithmic scale in Figure 5). This efficiency enables our method to successfully process all benchmarks, whereas Compigra suffers from timeouts on larger kernels such as **hotspot3D** and **lud**.

To further analyze these timing components, Figure 6 illustrates compilation time as a function of the number of operations per basic block. The ILP model

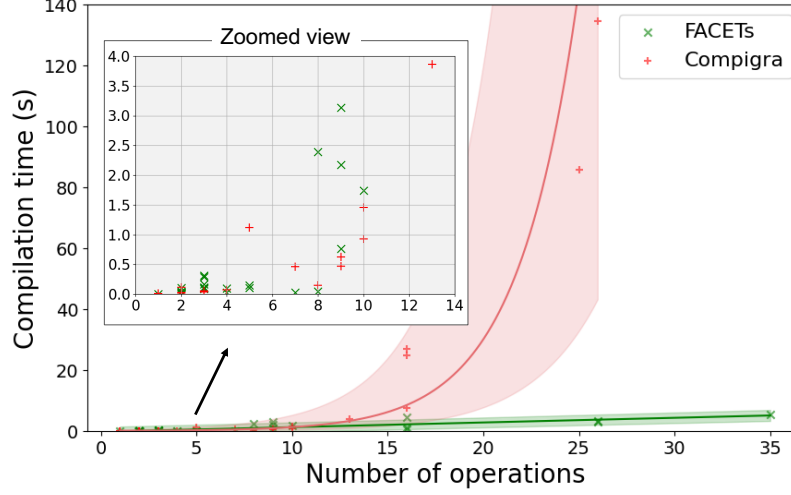


Fig. 6: Compilation time VS number of operations per DFG.

in Compigra exhibits exponential growth ($R^2 = 0.9210$), driven by the rapid increase in constraints as the number of operations increases. Conversely, our heuristic approach scales gracefully, with small increases in compile time due to more complex data dependencies for larger benchmarks. While Compigra is marginally faster (by only a few seconds) for small-scale problems, for higher number of operations the ILP model’s compilation time is orders of magnitude higher than ours — a critical bottleneck that makes it less practical for complex CDFG applications.

5.3 Runtime analysis

To assess the quality of FACETs schedules, we compared the achieved result with the lower bound derived from data dependencies and hardware constraints. The lower bound is defined by:

$$L_{\text{non-loop}} = \max(l_d, l_h); II_{\text{loop}} = \max(ResII, RecII)$$

For a non-loop basic block, the latency lower bound L is determined by the maximum of the data-dependency bound l_d and the hardware bound l_h . The data-dependency bound l_d corresponds to the length of the critical path (i.e., the longest data-dependency path) in the corresponding DFG, while the hardware-constraint bound l_h is computed as $N_{\text{op}}/N_{\text{PE}}$. For a loop basic block, the runtime lower bound is instead determined by the initiation interval II , which represents the minimum gap between the start of successive iterations. The II is constrained by both loop-carried data dependencies (captured by $RecII$) and the availability of hardware resources (captured by $ResII$) [4].

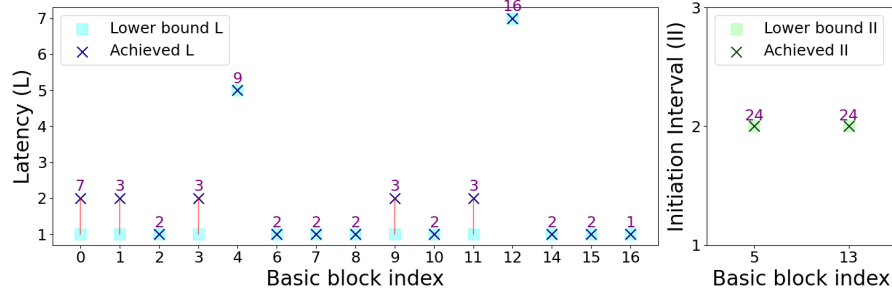


Fig. 7: Runtime analysis of `2mm` for nonloop blocks (left) and loop blocks (right), showing the blocks latency and initiation interval, respectively for the two cases. The $\times$ indicates achieved scheduling, $\square$ indicates the scheduling lower bounds. The number of operations per BB is noted above each data point.

Figure 7 shows the lower bound and implemented scheduling metrics of `2mm`, which shows the largest BB and operation counts. The achieved schedules are close to the runtime lower bounds in most cases, resulting in small gaps despite the large variation in operation counts. This indicates that the proposed scheduling approach efficiently exploits available parallelism and approaches the optimal scheduling limit under realistic resource constraints for `2mm`, which has a complex CFG structure. For performance-critical basic blocks, such as BB5 and BB13, which correspond to the innermost loops, DFG optimizations are applied first to simplify CFG constraints. The remaining basic blocks are then mapped while enforcing the required data-propagation patterns across the CFG. Basic blocks that exhibit gaps to the theoretical bounds are executed only once for data initialization or for data propagation between sequential loops, or they belong to outer loops with significantly fewer executions. Consequently, their scheduling overhead has a limited impact on overall runtime performance.

5.4 Runtime comparison with SoA methods

Figure 8 presents the runtime performance normalized to the baseline configuration, in which no MS is applied. Across most benchmarks, enabling DFG optimization achieves more than a $2\times$ speedup. This improvement is attributed to MS attaining an initiation interval [4] ($II < \frac{1}{2}L$), where L denotes the length of the longest loop-carried data dependency, thereby enabling effective loop folding. Consequently, each loop iteration completes in fewer cycles than in the baseline execution, which requires at least L cycles per iteration, resulting in substantially improved basic-block execution efficiency. The flexibility of integrating fast compilers for non-critical blocks and highly optimizing (slower) ones for critical ones is therefore key for achieving high runtime performance while targeting complex CFGs.

FACETs with MS consistently outperforms Marionette on benchmarks with deep loop nesting, such as `2mm` and `hotspot3D`. Marionette incurs runtime over-

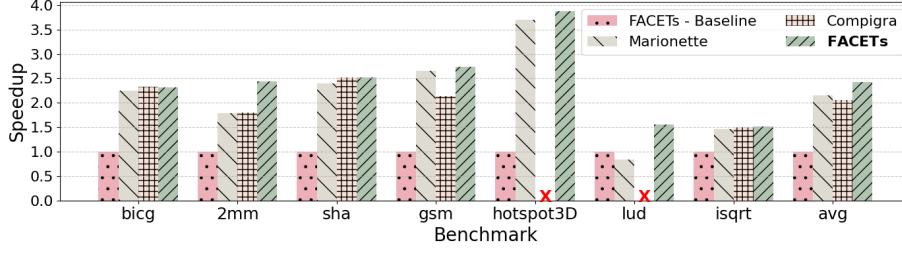


Fig. 8: Runtime comparison with state-of-the-art.

head due to data reconfiguration during basic-block switching, which becomes a performance bottleneck in nested-loop scenarios. In particular, for a three-level nested loop, the innermost loop requires reconfiguration $N_0 \times N_1$ times, where N_i denotes the iteration count at loop level i . This frequent reconfiguration significantly degrades runtime performance. For `lud`, Marionette fails to achieve speedup over the baseline because the innermost loop has a dynamic bound and the second-level loop has a dynamic start value, which prevents the scheduler from constructing a schedule with $II < L$.

Compared to Compigra, our approach slightly underperforms only on `bicg`, where Compigra’s ILP formulation is able to find an optimal placement within a reasonable solving time. However, when the optimal solution exceeds the ILP time limit, Compigra resorts to splitting the DFG via load/store operations, introducing additional overhead. In the worst cases, such as `hotspot3D` and `lud`, Compigra fails to produce a result within a reasonable time, as reflected by the missing data points in the figure.

6 Conclusion

We presented FACETs, a fast control-dataflow mapping framework for CGRAs that broadens the scope of compilation for applications with complex control flow. By combining heuristic mapping for non-critical regions with targeted optimization of performance-critical code under explicit control-flow constraints, the proposed approach significantly reduces compilation time while preserving execution efficiency. FACETs ultimately enables the acceleration of complex CD-FGs as single CGRA functions, minimizing costly host/CGRA data and configuration transfers overhead. Experimental results demonstrate up to $10\times$ faster compilation with comparable or improved runtime performance over existing methods.

Acknowledgments

This work was supported in part by the the Swiss NSF Edge-Companions project (GA No. 10002812) and the Swiss NSF grant no. 200021E_220194: “Sustainable and Energy Aware Methods for SKA (SEAMS)”; in part by the EC Horizon project CERBERUS under Grant 101223271; in part by the ACCESS—AI Chip

Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong, SAR; in part by the Swiss National Science Foundation via project ADApprox (grant 200020_188613); and in part by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips Research Project.

References

1. Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In Pedro C. Diniz, Eduardo Marques, Koen Bertels, Marcio Merino Fernandes, and João M. P. Cardoso, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, pages 1–13, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
2. Anish Vipplerla, Narayanan Murugan, Ali Akoglu, and Chaitali Chakrabarti. Compilation framework for dynamically reconfigurable array architectures. *IEEE Access*, 13:196415–196432, 2025.
3. Yuxuan Wang, Cristian Tirelli, Lara Orlandic, Juan Sapriza, Rubén Rodríguez Álvarez, Giovanni Ansaloni, Laura Pozzi, and David Atienza. An mlir-based compilation framework for cgra application deployment. In *International Symposium on Applied Reconfigurable Computing*, pages 33–50. Springer Nature Switzerland Cham, 2025.
4. B Ramakrishna Rau. Iterative Modulo Scheduling. 24(1):3–64, 1996.
5. Georgios Zacharopoulos, Lorenzo Ferretti, Emanuele Giaquinta, Giovanni Ansaloni, and Laura Pozzi. Regionseeker: Automatically identifying and selecting accelerators from application source code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):741–754, 2019.
6. Jinyi Deng, Xinru Tang, Jiahao Zhang, Yuxuan Li, Linyun Zhang, Boxiao Han, Hongjun He, Fengbin Tu, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. Towards efficient control flow handling in spatial architecture via architecting the control flow plane. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’23, page 1395–1408, New York, NY, USA, 2023. Association for Computing Machinery.
7. Satyajit Das, Kevin J. M. Martin, Davide Rossi, Philippe Coussy, and Luca Benini. An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(6):1095–1108, 2019.
8. Georgios Zacharopoulos and Laura Pozzi. Clrfreqcfigprinter: A tool for frequency annotated control flow graph generation. In *Eur. LLVM Developers Meeting*, 2017.
9. Yijiang Guo, Jiarui Wang, Jiaxi Zhang, and Guojie Luo. Formulating data-arrival synchronizers in integer linear programming for cgra mapping. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 943–948, 2021.
10. Alexander Chin and Jason Anderson. An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping. In *2018 55th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018.
11. Caleb Donovan, Makai Mann, Clark Barrett, and Pat Hanrahan. Agile smt-based mapping for cgras with restricted routing networks. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.
12. Cristian Tirelli, Juan Sapriza, Rubén Rodríguez Álvarez, Lorenzo Ferretti, Benoît Denkinger, Giovanni Ansaloni, José Miranda Calero, David Atienza, and Laura

- Pozzi. Sat-based exact modulo scheduling mapping for resource-constrained cgras. *J. Emerg. Technol. Comput. Syst.*, 20(3), aug 2024.
13. Cristian Tirelli, Lorenzo Ferretti, and Laura Pozzi. SAT-MapIt: A SAT-based modulo scheduling mapper for coarse grain reconfigurable architectures. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.
14. Cristian Tirelli, Lorenzo Ferretti, and Laura Pozzi. SAT-MapIt: An open source modulo scheduling mapper for coarse grain reconfigurable architectures. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 383–384, 2023.
15. Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Shiuan Peh, and Tulika Mitra. Morpher: An open-source integrated compilation and simulation framework for cgra. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*, 2022.
16. Dajiang Liu, Yuxin Xia, Jiaxing Shang, Jiang Zhong, Peng Ouyang, and Shouyi Yin. E2EMap: End-to-End Reinforcement Learning for CGRA Compilation via Reverse Mapping. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 46–60, March 2024.
17. Fábio T. Ramos, Pedro E. F. Realino, Wagner A. Junior, Alex B. Vieira, Ricardo S. Ferreira, and José Augusto M. Nacif. Smartmap: Architecture-agnostic cgra mapping using graph traversal and reinforcement learning. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025.
18. Yudong Mu, Siyi Li, Zhihua Fan, Wenming Li, Xuejun An, and Xiaochun Ye. Nfmap: Node fusion optimization for efficient cgra mapping with reinforcement learning. In Chao Li, Xuehai Qian, Dimitris Gizopoulos, and Boris Grot, editors, *Advanced Parallel Processing Technologies*, pages 61–73, Singapore, 2026. Springer Nature Singapore.
19. Shangli Li, Mingjie Xing, and Yanjun Wu. Hdcc: A hierarchical dataflow-oriented cgra compiler for complex applications. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference, ASPDAC '25*, page 265–271, New York, NY, USA, 2025. Association for Computing Machinery.
20. Heng Cao, Zhipeng Wu, Dejian Li, Peiguang Jing, Sio Hang Pun, and Yu Liu. Accelerating control flow on cgras via speculative iteration execution. *IEEE Computer Architecture Letters*, 24(1):109–112, 2025.
21. Jiahang Lou, Qilong Zhu, Yuan Dai, Zewei Zhong, Wenbo Yin, and Lingli Wang. Adora compiler: End-to-end optimization for high-efficiency dataflow acceleration and task pipelining on cgras. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7, 2025.
22. Yuxuan Wang, Cristian Tirelli, Giovanni Ansaloni, Laura Pozzi, and David Atienza. An mlir-based compilation framework for control flow management on coarse grained reconfigurable arrays. *arXiv preprint arXiv:2508.02167*, 2025.
23. M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001.
24. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

25. Cristian Tirelli, Rodrigo Otoni, and Laura Pozzi. Monomorphism-based cgra mapping via space and time decoupling. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7. IEEE, 2025.
26. Rubén Rodríguez Álvarez, Benoît Denkinger, Juan Sapriza, José Miranda Calero, Giovanni Ansaloni, and David Atienza Alonso. An open-hardware coarse-grained reconfigurable array for edge computing. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 391–392, 2023.