Prebypass: Software Register File Bypassing for Reduced Interconnection Architectures

Kanishkan Vadivel, Barry de Bruin, Roel Jordans, Henk Corporaal
Eindhoven University of Technology, The Netherlands

Pekka Jääskeläinen
Tampere University, Tampere, Finland

***Abstract*—Exposed Datapath Architectures (EDPAs) with aggressively pruned data-path connectivity, where not all function units in the design have connections to a centralized register file, are promising solutions for energy-efficient computation. A direct bypassing of data between function units without temporary copies to the register file is a prime optimization for programming such architectures. However, traditional compiler frameworks, such as LLVM, assume function-units connect to register-files and allocate all live variables in register-files. This leads to schedule inefficiencies in terms of instruction-level parallelism and register accesses in the EDPAs. To address these inefficiencies, we propose *Prebypass*; a new optimization pass for EDPA compiler backends. Experimental results on an EDPA class of architecture, Transport-Triggered Architecture, show that *Prebypass* improves the runtime, register reads, and register writes up to 16%, 26%, and 37% respectively, when the datapath is extremely pruned. Evaluation in a 28-nm FDSOI technology reveals that *Prebypass* improves the core-level Energy by 17.5% over the current heuristic scheduler.**

***Index Terms*—exposed datapath, TTA, LLVM, code generation.**

I. Introduction

The edge computing paradigm, where small low-cost and low-power embedded System-on-Chips (SoCs) are used for real-time on-device signal analysis, is gaining much more attention in the Internet-of-Things (IoT) field, especially in the remote healthcare, sensing, and control domains. Applications in these domains often require real-time operation and operate on confidential (user) data with limited energy budget. In these cases, on-device signal analysis (or edge processing) and actuation is required. When designing application specific instruction set processors (ASIP) for such applications, among other things, energy-efficiency, high performance and flexibility in reprogramming are prime requirements. Exposed datapath-architectures (EDPA) [1] are a promising solution, as they offer very high energy efficiency without compromising programmability.

As the name states, the EDPAs expose their datapath to the programmer. Exposing the datapath to the programmer permits programme-controlled register-file bypassing, referred to as *software-bypassing*, of data between Function-Units (FUs) at compile-time without the intervention of a Register-File (RF). Therefore, their RFs and datapath can be simplified without compromising performance. Various Coarse-Grained Reconfigurable Architectures (CGRAs) [2], Transport Triggered Architecture (TTAs) [3], Silicon-Hive's DSPs, TRIPS [4], and Explicit-SIMD [5] are examples of architectures that belong to this category. To extract most implementation benefits from EDPA, the datapath is typically pruned such that not all FUs have direct connectivity to a RF. Though this simplifies the hardware, the added additional compile-time responsibilities of assigning operations to FUs without datapath and RF port conflict increases compiler complexity [6], [7].

Most EDPA compilers [8]–[11] are designed with the LLVM framework [12] such that they benefit from language support (*clang*) and common optimisations for free. However, LLVM lacks support for modelling FU port buffers and a partially connected datapath between FUs and the RF. These limitations lead to inefficiencies for EDPA targets. Specifically, the register-allocation stage in the LLVM framework inherently assumes that all FUs have sufficient connectivity to a RF for operands and results, and allocates all the live variables in the RF. This worst-case register allocation leads to high register pressure and inefficient schedules on EDPA targets. Furthermore, for extremely pruned datapaths, where the RF is not reachable by all FUs, the scheduler forces data to the RF by passing the data values through other units, which leads to inefficient schedules. These shortcomings in existing compiler infrastructure motivate this work.

In this work we propose a compiler optimization algorithm for EDPA compiler targets, named $Prebypass$, to identify and skip register allocation for the live variables that are expected to be software-bypassed when scheduled. We make the following contributions:

- A novel $Prebypass$ algorithm for EDPA compilers to identify and skip register allocation for the live variables that will be software-bypassed when scheduled (Section IV).
- Exploration of the $Prebypass$ design parameters (node order heuristics and prebypass region size) to identify trade-offs in the algorithm configuration (Section VI-A).
- Evaluation of the $Prebypass$ optimization on an EDPA target in terms of runtime, register reads, register writes, and core level energy for a set of common signal processing kernels. Results show a decrease of *energy-delay-area product (EDAP)* of up to 13% (Section VI-B).

The proposed optimization is applicable to many EDPAs, such as CGRAs [2], that rely on an LLVM based compiler flow [7]. In this work we use the TCE toolset, an open-source (http://openasip.org) framework for TTA design and compilation, to demonstrate *Prebypass*. We selected the TTA

```
r1 = Load #10
r3 = r1 * r2
```

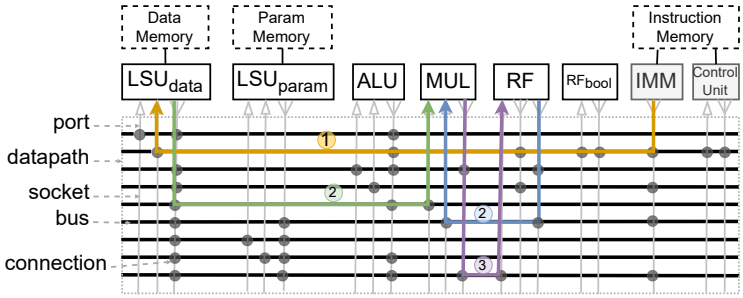
i) Sample program

```
IMM #10->Load.in
Load.out->RF.r1
RF.r1->MUL.1
RF.r2->MUL.2
MUL.out->RF.r3
```

ii) Program in TTA move form

```
Cycle-1: IMM #10->Load.in
Cycle-2: Load.out->MUL.1, RF.r2->MUL.2
Cycle-3: MUL.out->RF.r3
```

iii) TTA schedule with software bypassing



iv) TTA architecture instance

Fig. 1: Illustration of TTA programming model and architecture instance with visualization on how a program is decomposed into datapath *moves* at cycle level. The *moves* are numbered with their corresponding execution cycle. The flow dependency in register $r1$ is optimized out in the final program as a result of *software bypassing*.

processor as a representative EDPA class processor with an aim to open source the $Prebypass$ algorithm.

The remainder of this paper is organized as follows: Related works are discussed in Section II. Section III gives an overview of the TTA processor architecture and its compiler support. The proposed $Prebypass$ optimization algorithm and its results are outlined in Sections IV and VI, respectively. Section V outlines our experimental setup. Finally, Section VII describes future work and concludes the paper.

II. Related Works

Software bypassing is a key optimization that EDPAs rely on for their energy efficiency benefits [13]. The TCE compiler [14], [15], a state-of-the-art TTA compiler to our best knowledge, exploits bypassing opportunities primarily in the scheduling stage. The TCE scheduler attempts to schedule the operand and the result moves with software bypassing first. The scheduler falls back to register access for data when bypassing fails because of resource or dependency conflicts. The downside of the approach is that the bypassing is an optional optimization in the scheduling stage, and they are exploited only when the scheduling order of the nodes enforced by the scheduler presents an opportunity for bypassing.

The bypassing algorithms proposed for the Blocks CGRA and various TTAs in [8], [13], [16]–[18] follow a similar bypassing model where the scheduling stage exploits bypasses. In [8], [13]–[18], bypassing is not actively performed, but only when an opportunity arises. Registers are first allocated for all live variables and their use is optimised with bypasses in the scheduling stage. This over-allocation of physical registers leads to increased register pressure and often introduces false dependencies that restrict ILP in the program.

The scheduling algorithms based on integer linear programming [19] results in aggressive bypassing since they are immune to the scheduling order of the nodes. However, their use is limited to very small programs because of their extremely long scheduling time. The combined register allocation and scheduling [20] is another approach that helps with aggressive bypassing. The engineering complexity involved in their design limits their use.

The *safe-bypassing* approach presented in [21] is closely related to the proposed approach, and targets the same goal of improving software bypassing without over-allocating registers first. However, their resource model does not consider datapath availability and storage buffers in the input and output ports of the FUs. Furthermore, the approach is limited to processor architectures with single-cycle operations and FUs with a single output port. This restricts their use to a limited set of processor instances. Besides that, their approach is theoretical, and lacks implementation and validation.

III. Background

In this section we discuss the relevant background with respect to the TTA target (Section III-A) and the relevant compiler concepts (Section III-B).

A. Transport-Triggered Architectures

Transport Triggered Architecture (TTA) is a class of VLIW architecture with an exposed datapath. The TTA paradigm was proposed as an improvement to VLIW processors that suffer from register-file and bypass network scalability issues [3]. Like parallel operations in VLIW, the TTA instructions are composed of parallel moves that transport data between any two FU or RF ports in the design. TTA moves are fine-grained, each transferring a single operand. An operation is started on an FU when its final operand is moved to a special input port marked as trigger-input. Since the operands can be directly transferred between FU ports in a TTA (*software bypassing*), RF access is not necessarily needed, similar to traditional hardware bypasses in VLIW, but without complex hardware bypassing logic. Fig. 1 depicts a sample TTA instance and its programming model with software bypassing. Software bypassing reduces RF dependencies in instruction scheduling. Hence, the RF and datapath of a TTA instance can be simplified to match the application requirements. This enables TTA cores to be more scalable and energy-efficient compared to their VLIW counterparts.

The datapath in the TTA can be pruned in different ways, which we refer to as a connection model. The trivial connection model is the *fully connected* model, where all FU and

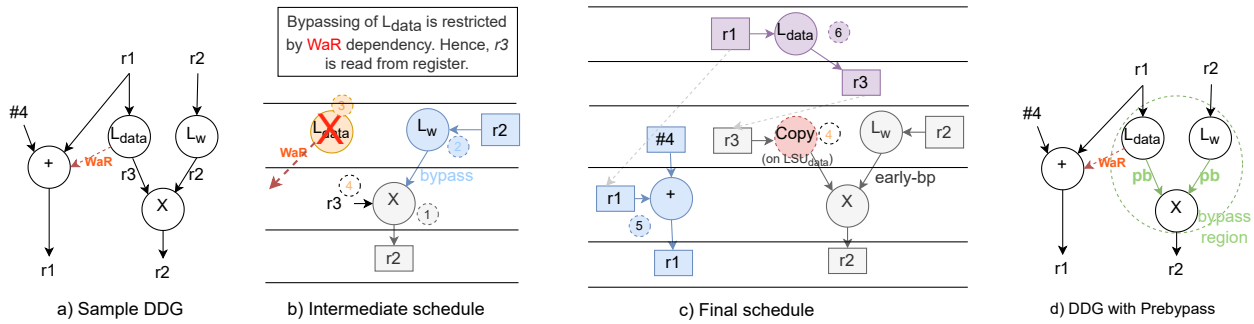


Fig. 2: Scheduler view of software bypassing support in TCE. The scheduling steps are colour-coded and numbered for interpretation: a) Sample DDG for schedule. b) The scheduling starts with the critical node (step-1). The input operands are attempted with software bypassing. If bypassing succeeds (step-2), the algorithm proceeds with the next move. Otherwise, the operand is read from RF (step-3 and 4) as a fall-back. This increases RF dependency in the schedule. c) When the FU lacks direct connectivity to RF, the data is routed via other units (step-4) by inserting a copy operation. This delays the schedule of L_{data} because of resource conflict.

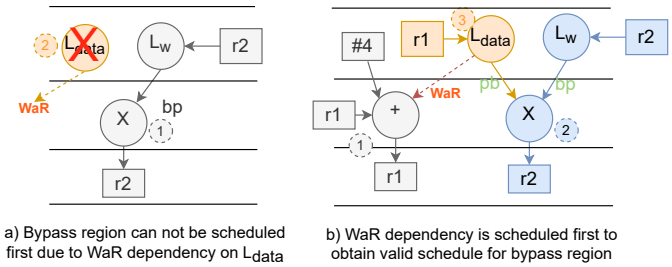


Fig. 3: *Prebypass* example. The scheduling steps are colour coded and numbered for interpretation. Schedule corresponds to DDG presented in Fig. 2d. a) When a schedule of a node fails in the bypass region, all the nodes that belong to the region are unscheduled and rescheduled later or in a different cycle. This enforces priority for bypassing over scheduler node ordering. b) The bypass region schedule is feasible when the scheduler attempts to schedule the bypass region in a different order.

RF ports are connected. The *directly reachable* model has at least one direct connection from each FU and RF output to all inputs. In a *fully RF connected* model, all FUs have connectivity to RFs but may miss connections between FU ports. And finally, the *disconnected RF* model, where some FUs are not directly connected to RF. The *disconnected RF* models use simpler hardware, but they are difficult targets for the compilers since they route register access of RF disconnected FUs via other units. Efficient compiler support for such architectures is still lacking, which restricts their use in practice.

B. TCE Toolchain and TTA Compiler

TTA-based Co-Design Environment (TCE) is an open-source toolset that enables the design and programming of customised ASIPs based on TTAs. The TTA compiler that is part of the tooling is based on the LLVM framework with TTA target as its backend. To overcome the shortcomings of LLVM in modelling datapath and FU ports, the backend implements a custom scheduling pass together with its own TTA resource model. The backend pipeline follows register-allocation first and then scheduling since spill code insertion after scheduling is not always possible in TTA [22], [23].

Exploiting bypass opportunities in the program is a prime optimization that enables EDPAs to reach high energy efficiency promises. In the current compiler flow, the scheduler identifies and extracts bypassing opportunities. Fig. 2 illustrates the bypass optimization in the scheduler stage. The scheduling step starts by selecting a node that is ready to be scheduled. The nodes in the critical path on Data Dependency Graph (DDG) are prioritised for being scheduled first since they dominate the schedule length. As shown in Fig. 2b (step 1), the *result-move* of the operation is scheduled first. Then their *operand-moves* are attempted to schedule with software bypass. The scheduling process proceeds with the next step if the software bypass succeeds (step 2 in Fig. 2b). If it fails, because of dependency or resource conflicts, the scheduler falls back to RF access for the operand (step 3 and 4 in Fig. 2b). However, this register access could be avoided when the "*Add*" node is prioritised to scheduled first. Furthermore, as shown in Fig. 2c, this fall-back register read is routed via other units when the RF is not directly reachable by the FU where the *result-move* is scheduled. This occupies additional resources and increases schedule length in the *disconnected RF* architecture.

This scheduling inefficiency occurs due to two key problems. First, bypassing is not a primary optimization in the scheduling stage and is exploited only when the node ordering enforced by the scheduler presents a bypass opportunity. Second, allocating registers for live variables that will be software bypassed leads to increased register pressure and often introduces false dependencies that restrict instruction-level parallelism (ILP) in the program. The proposed *Prebypass* algorithm addresses these two shortcomings by enforcing bypasses before register allocation.

IV. Prebypass Optimization

The goal of the *Prebypass* optimization is to improve bypassing capabilities of the post-RA scheduling model, where register allocation precedes the scheduling stage, without increasing the complexity of the EDPA scheduler. This is realised by implementing a pre-register allocation optimization pass that identifies the operand moves that should be bypassed when scheduled. The bypass candidates are identified using DDG analysis at the basic block level for a producer-consumer relation, and cross-checking target resource model for the availability of FU and datapath for software-bypassing the edge.

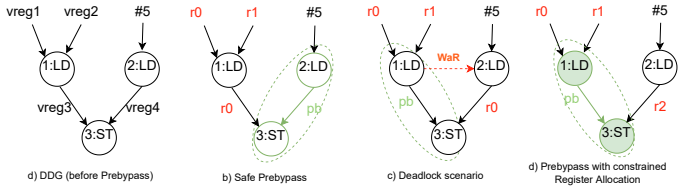


Fig. 4: An example of a deadlock situation in Prebypass. When the bypass cluster size is limited to one RaW, the pre-bypassing options are (b) and (c). Reusing $r0$ in (c) leads to cyclic-dependency for the bypass region, which results in a deadlock. Forcing register allocator to allocate different register for $vreg1$, $vreg2$, and $vreg4$ solves this issue as shown in (d).

Algorithm 1 Prebypass Algorithm

```
1:  procedure PREBYPASS(MachineBasicBlock mbb)
2:      ddg ← buildDDG(mbb)
3:      cand ← ddg.bypassCandidates()
4:      queue ← sort(cand, #orderingAlgorithm)
5:      for n ∈ queue do
6:          dist ← distance(n.source, n.destination)
7:          if dist≤TH[1] & machineHasResource(n) then
8:              Prebypass(n)
9:              if !deadlockFree(mbb) ||
10:                 bypassRegionSize(n) > TH[1] then
11:                 revertBypass(n)
12:             else
13:                 addRegisterAlloctionConstraints(n)
```

[1] *TH - User defined threshold.*

After identifying the bypass candidates, i.e, virtual registers holding live-variables, the algorithm marks them as pre-bypassed registers. The proceeding register allocation stage then skips register allocation for them. This reduces the number of physical register allocations and the accompanying Write-after-Read (*WaR*) and Write-after-Write (*WaW*) dependencies. Finally, $Prebypass$ hints the scheduler about the bypass opportunities. The scheduler then prioritises the $Prebypass$ nodes over its node ordering heuristics to exploit bypasses that scheduler bypasses fail to detect. The proposed $Prebypass$ optimization complements the existing scheduler bypass in improving bypassing capabilities of the compiler.

The rest of the section presents the extensions added to the current TTA compiler to support scheduling with pre-bypassing. Section IV-A presents the extension added to the scheduling stage. A model for pre-bypassing without introducing cyclic dependencies in the DDG is discussed in Section IV-B. Finally, Section IV-C presents the implemented $Prebypass$ algorithm.

A. Scheduling with Prebypass

$Prebypass$ forces flow dependencies, so called Read-after-Writes (*RaW*), to be a software bypass. The flow dependency is a producer to consumer relation in the data dependency, where the data produced by producer p is consumed by c. For an efficient schedule, the cycle in which the producer of the flow dependency can be scheduled is $cycle(p) = cycle(c) - Latency(p)$. Scheduling p later leads to longer resource occupancy and inefficient schedules. Scheduling earlier violates the RaW dependency. Hence, it is logical to schedule the pre-bypassed nodes in their optimal cycles. We implement a new scheduling model to achieve this with a more scalable solution by: a) Grouping the pre-bypassed nodes to form a cluster named *bypass regions*, and b) Scheduling the nodes in the bypass regions together. When the nodes are scheduled together, they can be placed at their optimal flow dependency cycle.

Fig. 3 explains the scheduling model for sample DDG with pre-bypassed nodes. The scheduling process starts with forming bypass regions by clustering the pre-bypass edges as shown in Fig. 2d. The scheduling heuristics then selects operations with its default critical path first node ordering and attempts to schedule the result move, followed by input operands with priority to the pre-bypassed operands. When scheduling pre-bypassed operands, the scheduler attempts to enforce software bypassing for the move as shown Fig. 3a. The heuristics proceed with the next operation when the schedule succeeds for all moves of the selected operation. Otherwise, scheduled moves of the selected operation are unscheduled. If the operation belongs to the bypass region, then all moves that belong to the region are unscheduled and scheduling is re-attempted in an earlier cycle or with a different operation, as shown in Fig. 3b. This ensures that the bypass regions are scheduled together for an efficient schedule.

B. Dependence Conflicts and Live-range Extension Problem

The bypass regions are individually scheduled to obtain resource and runtime efficient schedules. However, this constraint leads to schedule $deadlocks$ when there is a cyclic dependency between the bypass region and the regular node. The main source of this dependency is the register allocation stage which follows the *Prebypass* optimization. Physical registers are typically reused for holding live variables. This introduces false dependencies (like *WaR*) in the schedule. An example instance of this is given in Fig. 4c. The false dependency from physical register reuse is addressed in our work by constraining the register allocator from reusing the same physical registers for problematic edges in the DDG. For the example in Fig. 4a, by enforcing the register allocator to assign independent registers for *vreg1, vreg2, and vreg4*, the deadlock from false dependency can be avoided. The resulting DDG with register allocator constraint is given in Fig. 4d.

In the LLVM framework, this can be achieved by extending the live range of the conflicting nodes to overlap with each other. In some cases, this approach increases register pressure of the code block, which may result in an inefficient schedule. In our example (Fig. 4d), the register need increased from two registers to three. Our observations of this effect on real benchmark kernels are presented in Section VI-A. Furthermore, other optimizations such Phi-elimination and loop invariant code movement may also introduce dependencies due to code movement. Avoiding nodes such as *Phi* and *Copy* nodes, whose behaviour is not determined before register allocation, when selecting pre-bypassing candidates solves this issue.

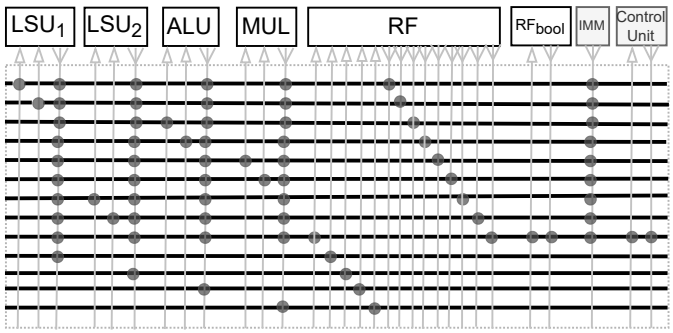


Fig. 5: *VLIW-TTA:* A *VLIW connectivity* TTA, where each FU port in the design is assigned with a dedicated register file port via a bus. The output ports connected to all input ports in the design are to allow for the maximum bypassing opportunity.

Kernel	Description	Inner-loop DDG		
		Node count	Nodes in critical path	DDG width/height
dotp	Dot product unrolled 8x	49	12	16/12
fir	Fully unrolled 8-tap FIR	35	15	9/12
gemm	Mat-Mul without unroll	8	5	4/5
iir	Fully unrolled 3rd order IIR	129	24	24/18
latsynth	Lattice synthesis	39	15	13/7
projection	Sums each row/column	24	6	7/6
conv	3x3 convolution unrolled to kernel width	33	11	24/18

TABLE I: Benchmark characteristics.

C. Prebypass Algorithm

The pseudocode presented in Algorithm 1 summarises the Prebypass algorithm. The algorithm operates at the basic-block level and has two main design parameters, 1) *Node ordering*, the order in which the edges are bypassed, and 2) *Bypass region size*, a limit on maximum number of bypassed edges in a $Prebypass$ cluster. The $Prebypass$ work alongside the TCE scheduler bypasses. Hence, pre-bypassing the nodes that are trivial to the scheduler bypassing is unnecessary and prioritising the nodes that are non-critical to the schedule leaves sub-optimal performance. On the other hand, including too many edges in the bypass region may cause a poor schedule since it delays the schedule of other nodes in the basic block. When restricted too much, schedule critical nodes loose pre-bypass opportunity. Therefore, the choice of node ordering and bypass region size plays a vital role in the Prebypass design. The results of these two parameter explorations are presented in Section VI-A.

The $Prebypass$ algorithm starts with building of DDG for the basic block. Then, it identifies the pre-bypass candidate by filtering out the nodes that might expand in a later stage of the compiler pipeline. The filtered candidates are then sorted with a defined node-ordering algorithm and are attempted to pre-bypass one by one.

In the pre-bypassing loop, the distance between the producer and consumer node is verified to be within the threshold. The LLVM arranges the instructions inside the basic blocks for optimal register live-range. The pre-bypass extends the live range of the producer and consumer operands to avoid deadlocks from false dependencies (Section IV-B). Hence, pre-bypassing nodes that are far apart in DDG leads to high-register pressure and possibly more false dependencies for other nodes in the basic block. If they are within the threshold, pre-bypass is attempted if the machine resources are available for bypassing the edge. The machine resource check validates the availability of sufficient FUs resources for computation, FU port buffers to hold operands and results, and connectivity in the datapath for bypassing. After bypassing, deadlock checks are performed and bypass region size is verified to be within the defined threshold. The register allocation constraints are added to the edge if the checks pass, otherwise, pre-bypass is reverted and the algorithm continues with the next candidate.

V. Experimental Setup

In order the evaluate the impact of the proposed pre-bypass optimizations, as proposed in Section IV, we have implemented the algorithms with LLVM 13.0 in the TCE framework [9]. We consider seven benchmarks (see Table I) of representative loop-oriented signal processing and deep learning kernels in our evaluation. We carefully chose the benchmarks to cover the different basic block properties encountered in practical applications. Specifically, the number of nodes in the basic-block (basic-block size), nodes in the critical path, and data-dependency characteristics (DDG height/width) are used to derive the benchmark set.

We consider three TTA cores with a varying number of RF ports and connectivity restrictions. The first core implements a traditional VLIW connectivity, where a dedicated RF port and a bus are assigned for each FU port in the design. Besides that, the connectivity model implements maximum bypassing

Benchmark	Metric	Node ordering (% difference over baseline)								
		A	B	C	D	E	F	G	H	I
dotp	reg-reads	50.0	57.6	42.3	42.3	42.3	38.4	38.4	42.3	53.8
	reg-writes	52.9	58.7	64.6	64.6	47.0	35.2	41.1	64.6	47.0
	runtime	24.2	26.9	10.8	8.1	16.2	16.2	16.2	8.1	24.2
fir	reg-reads	50.0	50.0	45.5	54.5	50.0	45.5	45.5	54.5	63.6
	reg-writes	68.7	68.7	62.5	68.7	68.7	62.5	62.5	68.7	68.7
	runtime	41.0	41.0	38.5	33.3	41.0	38.5	38.5	30.8	38.5
gemm	reg-reads	24.0	24.0	12.0	12.0	24.0	24.0	12.0	12.0	24.0
	reg-writes	38.2	37.9	19.1	19.1	19.1	37.9	19.1	19.1	37.9
	runtime	17.4	8.6	8.7	8.7	8.7	8.6	8.7	8.7	17.4
iir	reg-reads	6.7	6.7	-4.2	-3.4	-2.5	-2.5	-4.2	-5.9	8.4
	reg-writes	3.7	3.7	-11.0	-6.1	-9.8	-9.8	-11.0	-9.8	6.1
	runtime	3.9	3.9	-7.1	-7.9	-0.8	-0.8	-7.1	-3.1	3.1
latsynth	reg-reads	-3.7	-3.7	3.7	-3.7	-3.7	-3.7	-3.7	-3.7	-3.7
	reg-writes	0.0	0.0	9.4	0.0	0.0	0.0	0.0	0.0	9.4
	runtime	8.5	8.5	4.2	8.5	8.5	8.5	8.5	8.5	4.2
projection	reg-reads	7.9	7.9	7.9	7.9	7.9	7.9	7.9	7.9	7.9
	reg-writes	12.3	12.3	12.3	12.3	12.3	12.3	12.3	12.3	12.3
	runtime	20.8	20.8	20.8	11.9	20.8	20.8	20.8	11.9	11.9
conv	reg-reads	28.2	28.2	9.4	33.0	28.2	28.2	28.2	28.2	28.2
	reg-writes	30.4	30.4	15.2	38.0	38.0	38.0	30.4	38.0	30.4
	runtime	3.3	3.3	-3.3	0.0	3.3	3.3	3.3	3.3	3.3

Node orderings

A: criticalMulti_criticalSingle_Multi_Single, ***E***: criticalMulti_criticalSingle
B: criticalSingle_criticalMulti_Single_Multi, ***F***: criticalSingle_criticalMulti
C: criticalMulti_Multi, ***G***: criticalMulti, ***H***: criticalSingle,
D:criticalSingle_Single, ***I***: randem_order_1000samples

TABLE II: Comparison of different node ordering in terms of register-reads, register-writes, and runtime on a *restricted-TTA* with RF size of 16 (i.e 16x32-bit registers). To eliminate the influence of bypass region size, we compiled each benchmark with different bypass region sizes and selected the best runtime among them. The $green$ and red colour code highlights the max performance gains and all the worsening, respectively.

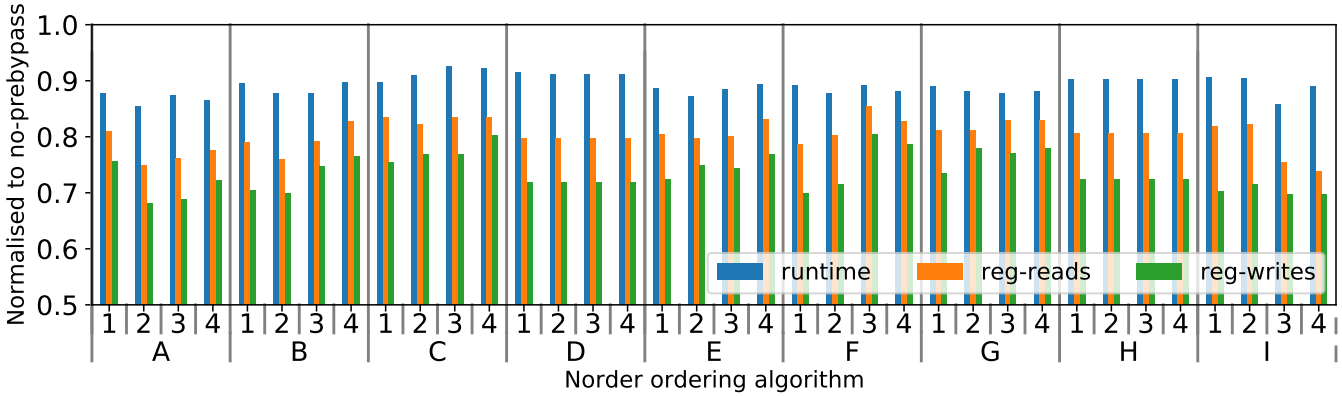


Fig. 6: Effect of region size on different node orderings in *restricted-TTA*. The values correspond to the average benefit for all the benchmarks.

support, where any FU can read the output of any other FU to mimic a realistic VLIW processor model. This enables this architecture to be free from connectivity or register-file port restrictions. Fig. 5 depicts the considered *VLIW-TTA* core.

We constrained the register-file ports in the VLIW connectivity to two read and one write port to derive the second configuration, named a register-port constrained TTA (*rc-TTA*). Finally, a TTA with *disconnected RF* connectivity, a *restricted-TTA*, is formed by pruning the connectivity network of *rc-TTA*. A greedy iterative connection pruner is used to prune connections in all buses with minimal impact on execution time. Fig. 1 (iv) shows the resultant *restricted-TTA* from the connection pruner.

For accurate area, energy, and runtime estimations, the cores are synthesized (using Cadence Genus) in a commercial 28-nm FD-SOI (Fully Depleted Silicon On Insulator) technology (SS corner), using a 12-track RVT standard cell library and Foundry SRAM memories. Power analysis is performed using back-annotated post-synthesis netlist simulations under typical operation conditions (25°C, 0.9 V, 800 MHz).

VI. Results

Explorations of the two Prebypass design parameters namely, a) *Node ordering* heuristics, and b) *Bypass region size*, are presented first in Section VI-A. Using the identified design parameter values, the Prebypass optimization is evaluated in Section VI-B.

A. Prebypass Design Parameter Exploration

The nodes in the DDG can either be a critical node that belongs to a critical path or a regular node. Nodes can be further distinguished based on the number of operands as single- or multi-operand nodes. This classification allows for different node orderings. To understand the trade-offs, the performance of these ordering is compared in terms of register-reads, register-write, and runtime for *restricted-TTA*. Table II summarises these results.

Pre-bypassing critical nodes: Pre-bypassing critical nodes of a basic block (orderings *G*, *H*, *E* and *F* in Table II) is a logical choice. However, the number of candidates available for bypassing is very limited in these orderings which leads to sub-optimal benefits for most benchmarks. The orderings *C* and *D* consider the entire node set in a basic block, and selects either single- or multi-operand nodes and prioritises critical nodes in the set. Depending on the ratio of the single- to multi-operand nodes, each benchmark favours one or the other order. However, the overall performance remains sub-optimal. The iir benchmark has a very high ILP, less number of critical nodes, and equal mix of single- and multi-operand nodes. Hence, the ILP dominates the critical path influence in the schedule. Therefore, pre-bypassing critical or single/multi-operand nodes restricts the bypassing opportunities of regular nodes and ILP, which results in more register accesses and an increase in runtime.

Aggressive Pre-bypassing: The orderings A and B, where all nodes are considered for pre-bypassing with critical path sorted manner, benefit the entire benchmark set and yields the best results. This is because the ordering presents more pre-bypass candidates, and hence the basic block is pre-bypassed aggressively. Prioritizing single- or multi-operand nodes did not influence the results, since aggressive bypassing improved the schedule of all the nodes in the basic blocks. The $gemm$ kernel runtime is an outlier to this. The bypass regions formed when multi-operands are prioritised (ordering A) favour the scheduling node selection heuristics and the scheduler bypassing. Therefore, the scheduler was able to achieve a spatial layout in the schedule with optimal runtime.

Bypass region size: Fig. 6 presents the effect of region size on different node orderings. When only single-operand nodes are considered for pre-bypassing (ordering D and H), the pre-bypassing opportunities are limited. The pre-bypassing opportunity saturates at bypass region with size one. Hence, increasing the region size beyond one has no effect on the performance. When multi-operand nodes are prioritised (orderings C, E, and G), the register pressure increases proportionally to the bypass region size. Hence, going for a higher bypass region size leads to fewer benefits. When single- and multi-operand nodes are mixed (orderings A, B, and F), the performance gain improves until bypass region size two and then decreases for higher region sizes. When the

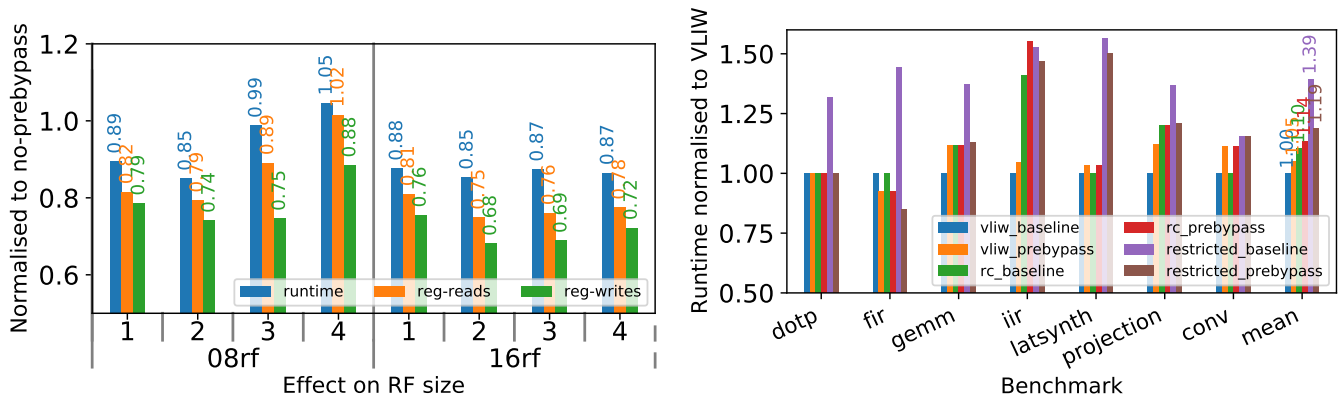


Fig. 7: Effect of register pressure on $Prebypass$ optimization benefits on *restricted-TTA* (left) and the runtime benefits of $Prebypass$ optimization on *VLIW-TTA*, *rc-TTA*, and *restricted-TTA* with RF size of 16 (right). $Prebypass$ reduces runtime in *restricted-TTA* and increases it on *VLIW-TTA* and *rc-TTA*.

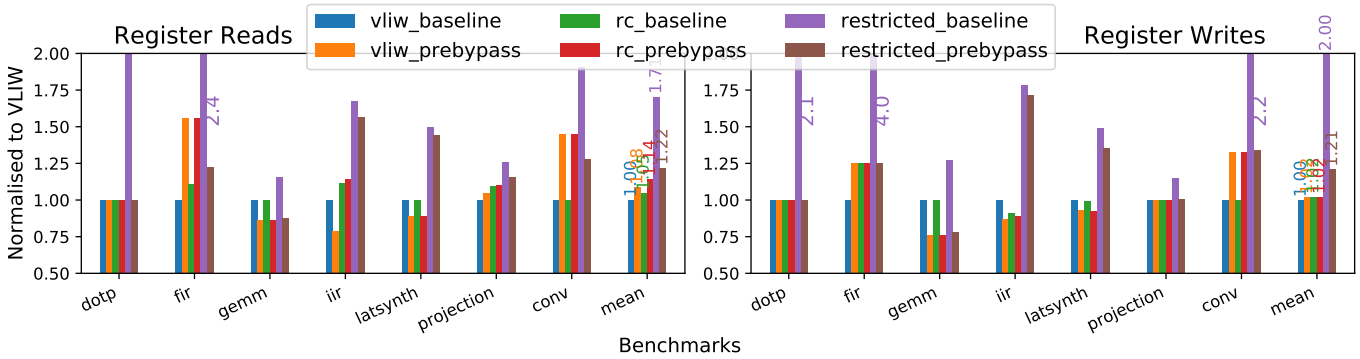


Fig. 8: The register-read and register-write benefits of $Prebypass$ optimization on *VLIW-TTA*, *rc-TTA*, and *restricted-TTA*. Prebypass reduces register-reads and register-writes in *restricted-TTA* and increases them on *VLIW-TTA* and *rc-TTA*.

region size is high, bypass regions occupy larger regions in the schedule. This worsens the schedule of the nodes outside of the bypass region. Therefore, a combination of bypass region size two with node ordering A achieves the best result for our benchmark set.

Fig. 7 presents the effect of pre-bypass region size under register pressure. We consider the *restricted-TTA* core with a varying number of RF sizes. Irrespective of the register pressure, the bypass region size two is optimal for the considered benchmarks. A larger bypass region size reduces performance, due to an increase in register pressure. The live-range extension from pre-bypass leads to more false dependencies and possibly creates spill code during register allocation.

B. Evaluation of Prebypass Optimization

This section presents the evaluation of Prebypass optimization on the three TTA architectures detailed in Section V. The *VLIW-TTA* is used as a baseline for our analysis. Fig. 7 (right) and Fig. 8 present the runtime, register-read, and register-write benefit of $Prebypass$ optimization over the baseline for the selected architectures. The bypass region size is set to two and node ordering algorithm A is selected for the analysis. The existing scheduler bypassing of the TCE compiler [14] is enabled in both baseline and pre-bypass versions.

For the *VLIW-TTA* there are no RF port and connectivity restrictions. As such, the scheduler can reach its best schedule for the FUs in this model. Enabling $Prebypass$ often makes it worse, since the bypass parameters (node ordering and region size) are tuned for *restricted-TTA*. Hence, forcing the pre-bypass harms most benchmarks. Re-tuning the $Prebypass$ parameters helps these cases, but we did not endeavour it since our focus is *restricted-TTA*. Hence, the average runtime, reg-reads, and register-writes increases by 5%, 8%, and 2% respectively. When the register file ports are constrained to two reads and one write port in *rc-TTA*, the kernels $dotp$, fir, $latsynth$, and $conv$ retain their performance since they are not constrained by RF bandwidth. The runtime of $gemm$, iir, and $projection$ kernels increases on the *rc-TTA*, while their register access remains the same. This indicates a RF bandwidth bottleneck for these kernels. Pre-bypassing has the same effect as *VLIW-TTA*, and average runtime and register-reads increases by up to 3.5% when pre-bypass is enforced.

Pruning connections in the architecture leads to a significant increase in runtime for all benchmarks because of, 1) when FUs are not directly connected to RF, the FUs access the register via other units, which increases schedule length. 2) Schedule length increases when connectivity in the datapath is not favouring the application requirements. $Prebypass$ recov-

Core	Config.	Max Freq. (MHz)	% difference over *baseline* Energy	% difference over *baseline* EDAP	% difference over *VLIW-TTA* Energy	% difference over *VLIW-TTA* Area	% difference over *VLIW-TTA* EDAP
VLIW-TTA	baseline [14]	800	-	-	-	-	-
	+ prebypass	800	3.1	3.1	3.1	-	3.1
rc-TTA	baseline [14]	800	-	-	-0.3	-14.4	-14.6
	+ prebypass	800	1.44	1.44	1.18	-14.4	-13.4
restricted-TTA	baseline [14]	800	-	-	15.6	-34.7	-24.5
	+ prebypass	800	**-17.5**	**-17.5**	**-4.7**	**-34.7**	**-37.8**

TABLE III: Energy, Delay, and Area estimations of *VLIW-TTA*, *rc-TTA*, and *restricted-TTA* cores (with RF size of 16)

ers the negative impact of connectivity restrictions completely in $dotp.\ gemm$ and $projection$ kernels by eliminating $copy$ nodes in the schedule. In the *restricted-TTA* architecture only nodes that match the data-path connectivity are schedulable at the earliest cycle. Therefore, a node that fits in the earliest cycle is chosen for scheduling. In the fir benchmark, this ordering favours the architecture connectivity model and hence it results in a better schedule than the baseline model. The $conv$, $latsynth$, and iir kernels have the least favourable connections. Hence, pre-bypassing reduces runtime, reg-reads, and register-writes up to by 14.3%, 28.6%, and 39.5% respectively in this connectivity model. Even though pre-bypassing improves their runtime and register accesses, their performance is behind the baseline by 20%. We observed negligible compile time implications when $Prebypass$ is used in the compiler pipeline. This observation emphasizes the scalability of the $Prebypass$ algorithm.

Table III presents the Energy, Delay, and Area estimations of *VLIW-TTA*, *rc-TTA*, and *restricted-TTA* cores from post-synthesis netlist simulations. For the considered architecture, the register file and datapath are not on the critical path, which explains the equivalent maximum clock frequency between all architectures. The *VLIW-TTA* architecture occupies the maximum area because of the datapath and RF complexity. The area footprint improves by 14% when the RF ports are pruned in *rc-TTA*. This improves the *Energy, Delay, and Area Product (EDAP)* of *rc-TTA* up to 14.6%. When pre-bypassing is enforced on *VLIW-TTA* and *rc-TTA*, the runtime increases and this directly translates to an EDAP increase. Pruning the datapath on top of the RF port constraint increases the energy per kernel by 15.6% because of the longer runtime and unnecessary $copy$ nodes in the schedule. However, the pre-bypassing improves runtime by up to 14.3%. Together with simplified hardware, the *restricted-TTA* architecture achieves EDAP gains of up to 37.8% over *VLIW-TTA* for our benchmarks.

VII. Conclusion

We presented a novel $Prebypass$ optimization for EDAP architectures to expose more software-bypassing opportunities to the compiler. The optimization improves the scheduling quality in pruned datapaths where not all FUs are connected to a single centralized RF. Compared to the state-of-the-art TTA compiler that optimizes software-bypassing after register allocation, adding the $Prebypass$ optimization improves the *average* runtime, register reads, and register writes by 16 %, 26 %, and 37 %, respectively. Evaluation in a 28-nm FDSOI technology indicates that pruning the number of RF ports leads to a 14.4 % area improvement without a significant impact on schedule quality. Pruning the data-path further improves the area reduction from 14.4 % to 34.7 %, at the cost of an *average* 15.6 % energy increase due to inefficient bypassing support in the compiler. The $Prebypass$ optimization exposes more bypass opportunities to the compiler, and improves the *average* energy from +15.6 % to -4.7 %, highlighting the importance of $prebypass$ on heavily pruned datapaths. In the future, we aim to integrate node ordering and bypass region size selection logic inside the algorithm and extend the scope to SIMD and highly parallel architectures with more FUs.

Acknowledgments

This work is part of the research programme Perspectief ZERO with project number P15-06 Project 5, which is (partly) financed by the Dutch Research Council (NWO). This work is also supported by HiPEAC collaboration grant and by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware).

References

[1] P. Jääskeläinen, "Code density and energy efficiency of exposed datapath architectures," *Journal of Signal Processing Systems*, 2015.
[2] M. Wijtvliet, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *SAMOS*, 2016.
[3] H. Corporaal, *Microprocessor Architectures: from VLIW to TTA*, 1997.
[4] K. Sankaralingam, R. Nagarajan, H. Liu, and Kim, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *ISCA*, 2003.
[5] L. Waeijen and D. She, "A low-energy wide simd architecture with explicit datapath," *Journal of Signal Processing Systems*, 2015.
[6] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of transport triggered architectures," in *MICRO*, 1994.
[7] K. Vadivel, R. Jordans, and S. Stujik, "Towards efficient code generation for exposed datapath architectures," ser. SCOPES '19, 2019.
[8] M. Adriaansen, M. Wijtvliet, and R. Jordans, "Code generation for reconfigurable explicit datapath architectures with llvm," in *DSD*, 2016.
[9] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*, 2017.
[10] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of cgra," in *FCCM*, 2014.
[11] V. Govindaraju, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, 2012.
[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004.
[13] V. Guzma, "Impact of software bypassing on instruction level parallelism and register file traffic," ser. SAMOS '08, 2008.
[14] H. O. Kultala and T. T. Viitanen, "Aggressively bypassing list scheduler for transport triggered architectures," in *SAMOS*, 2016.
[15] H. O. Kultala, P. O. Jääskeläinen, J. IJzerman, and L. K. Lehtonen, "Exposed datapath optimizations for loop scheduling," in *SAMOS*, 2017.
[16] V. Guzma, T. Pitkänen, and J. Takala, "Use of compiler optimization of software bypassing as a method to improve energy efficiency of exposed data path architectures," *EURASIP Journal on Embedded Systems*, 2013.
[17] D. She and Y. He, "Scheduling for register file energy minimization in explicit datapath architectures," in *DATE*, 2012.
[18] K. Vadivel, "Energy efficient loop mapping techniques for coarse-grained reconfigurable architecture, master's thesis,," 2017.
[19] T. Äijö and P. Jääskeläinen, "Integer linear programming-based scheduling for transport triggered architectures," *TACO*, 2015.
[20] J. Janssen, "Registers on demand, an integrated region scheduler and register allocator," in *Conference on Compiler Construction*, 1998.
[21] P. Kellomäki and V. Guzma, "Safe pre-pass software bypassing for transport triggered processors," *ATN*, 2008.
[22] J. A. A. J. Janssen, "Compiler strategies for transport-triggered architectures." *Delft University of Technology*, 2002.
[23] H. Corporaal, "Code generation for transport triggered architectures," in *Code Generation for Embedded Processors*, 2002.