Improving Utilization of Dataflow Architectures Through Software and Hardware Co-Design

Zhihua Fan[1,2], Wenming Li[1,2(✉)], Shengzhong Tang[1,2], Xuejun An[1,2], Xiaochun Ye[1,2], and Dongrui Fan[1,2]

[1] SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{fanzhihua,tangshengzhong20s,axj,yexiaochun,fandr}@ict.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
liwenming@ict.ac.cn

Abstract. Dataflow architectures can achieve much better performance and higher efficiency than general-purpose core, approaching the performance of a specialized design while retaining programmability. However, dataflow architectures often face challenges of low utilization of computational resources if the application algorithms are irregular. In this paper, we propose a software and hardware co-design technique that makes both regular and irregular applications efficient on dataflow architectures. First, we dispatch instructions between dataflow graph (DFG) nodes to ensure load balance. Second, we decouple threads within the DFG nodes into consecutive pipeline stages and provide architectural support. By time-multiplexing these stages on each PE, dataflow hardware can achieve much higher utilization and performance. We show that our method improves performance by gmean 2.55× (and up to 3.71×) over a conventional dataflow architecture (and by gmean 1.80× over Plasticine) on a variety of challenging applications.

Keywords: Dataflow Architecture · Decoupled Architecture

1 Introduction

Dataflow architecture is an emerging class of reconfigurable arrays designed for modern analytical workloads. The program offloaded to dataflow fabrics will be converted to a dataflow graph (DFG) by dataflow compiler. A DFG consists of a set of nodes and directed edges that connect the nodes. The nodes represent the computing, while the edges represent data dependencies between nodes. Figure 1 illustrates a typical dataflow architecture, which consists of a PE (Processing Element) array, a configuration buffer and a data buffer. The PE array is formed by multiple PEs that are connected by the network-on-chip. Each PE is composed of a router, a local buffer, a register file, and several function units.

https://doi.org/10.1007/978-3-031-39698-4_17

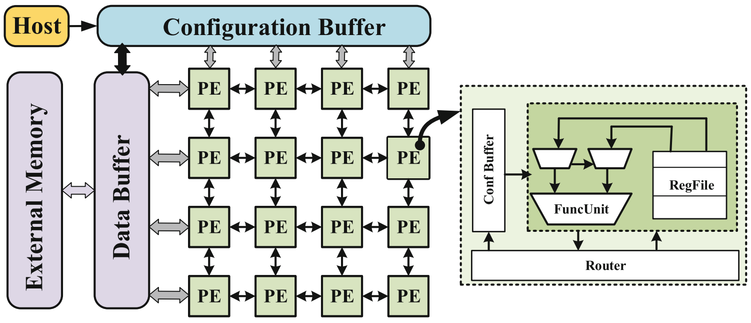


Fig. 1. A typical dataflow architecture.

Most dataflow architectures are restricted to regular applications, i.e., those with structured access patterns and data reuse, like neural networks [1] and dense linear algebra [2]. These characteristics are necessary to produce a high-performance pipeline that can be spatially and statically mapped to a dataflow fabric. However, dataflow architectures struggle with irregular applications, i.e., those with complex control flow and memory access patterns, lack data sharing characteristics and data reuse. These applications arise in many important domains, like graph analytic, sparse linear algebra and signal processing. Dataflow architectures are ill-equipped to handle these operations.

Abundant prior works have been proposed to accelerate irregular applications on dataflow architectures (in Sect. 2): pipeline parallelism [3–5], decoupled access-execute architectures [6–9] and dedicated interfaces between cores or threads [10,11]. Nevertheless, these solutions can be further improved because they (in Sect. 3): (i) suffer from load imbalance, as they rarely take into account the imbalance between DFG nodes, but we found the imbalance impacts the software pipeline execution significantly. (ii) lack of fine-grained pipelining scheduling. The schedule of each DFG node is coarse-grained and non-preemptive, which miss opportunities to exploit more parallelism within DFG nodes to boost utilization. To this end, we introduce a software and hardware co-design method to improve the hardware utilization of dataflow architectures. In summary, we make the following contributions:

- We present a method to solve the load imbalance between DFG nodes. This approach dispatches instructions between DFG nodes to ensure load balance.
- We introduce *decoupled execution model*. It decouples the thread within DFG node into four consecutive pipeline stages. Each stage is an atomic schedule and execution unit. In this way, a PE can be shared by at most four different DFG nodes at the same time and the memory access and data transfer latency can be overlapped as much as possible.
- We provide architectural support for the decoupled execution model. By decoupling the datapath of different stages and equipping with a dedicated scheduler within each PE, the DFG nodes of different iterations can be pipelined more efficiently.

– We evaluate our methods on a wide range of applications, demonstrating their applicability. Experiments show our methods improve performance by gmean 2.55× (and up to 3.71×) over a conventional dataflow architecture (and by gmean 1.80× over Plasticine [4]).

2 Background and Related Works

In this section, we briefly introduce the background and works related to improving the utilization of dataflow architectures.

Pipeline Parallelism: Dataflow architectures are amenable to creating static spatial pipelines, in which an application is split into DFG nodes and mapped to functional units across the fabric [1–4,12]. To perform a particular computation, operands are passed from one functional unit to the next in this fixed pipeline. Pipette [5] structures irregular applications as a pipeline of DFG nodes connected by queues. The queues hide latency when they allow producer nodes to run far ahead of consumers. Zhongyuan et al. [13] design a global synchronization mechanism, which help reducing the nodes and edges in modified DFG. and propose a complete and systematic DFG modification flow which saves more resources. These efforts may be inefficient for irregular workloads because they rarely take into account the load imbalance between DFG nodes.

Decoupled Architectures: Fifer [6] decouples memory access datapath from computing pipeline. Each DFG node is divided into two stages: access and execution. Equipped with a dedicated scheduler, at most two DFG nodes can be executed on the same PE at the same time. In this way, memory access latency can be overlapped and the utilization can be further improved. DESC [7] proposes a framework that has been inspired by decoupled access and execution, and updates and expands for modern, heterogeneous processors. REVEL [8] extends the traditional dataflow model with primitives for inductive data dependences and memory access patterns, and develops a hybrid spatial architecture combining systolic and dataflow execution. RAW [9] introduces hardware support for decoupled communication between cores, which can stream values over the network. Käsgen et al. [14] present a new mechanism that resolves data and structural hazards in processing elements that feature in-order issue, but out-of-order completion of operations. Different from these partial design, our methods is fully decoupled PE.

Custom Interface: Chen et al. [11] propose subgraph decoupling and rescheduling to accelerate irregular applications, which decouples the inconsistent regions into control-independent subgraphs. Each subgraph can be rescheduled with zero-cost context switching and parallelized to fully utilize the PE resources. TaskStream [10] introduces a task execution model which annotates task dependences with information sufficient to recover inter-task structure. It enables work-aware load balancing, recovery of pipelined inter-task dependences, and recovery of inter-task read sharing through multicasting. MANIC [15] introduces vector-dataflow execution, allowing it to exploit the dataflow in a sequence of vector instructions and amortize instruction fetch and decode over a whole vector of operations. By forwarding values from producers to consumers, MANIC avoids costly vector register file accesses. However, the schedule mechanism of DFG nodes within each PE is coarse-grained and non-preemptive. The PE can switch to the next iteration or other nodes only after finishing all instructions of the current DFG node.

3 Motivation

Irregular applications are common in real-life workloads and benchmark suites, such as graph analytics, sparse linear algebra and databases. As reported in Fig. 2 (a), the average percentage of unstructured access, complex control flow and imperfect loops can be over 50% in three widely-used benchmarks. Figure 2 (b) reports the utilization of the dataflow fabrics using the methods we discussed earlier (Sect. 2). Obviously, the hardware utilization is pretty low and at least half of the PEs are under-utilized during execution. We obtain these results from experiments with a dataflow simulator, using the methods introduced in [4] and [6], respectively.

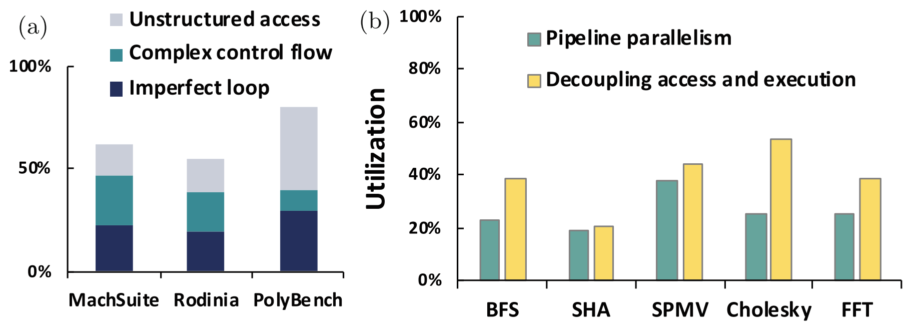
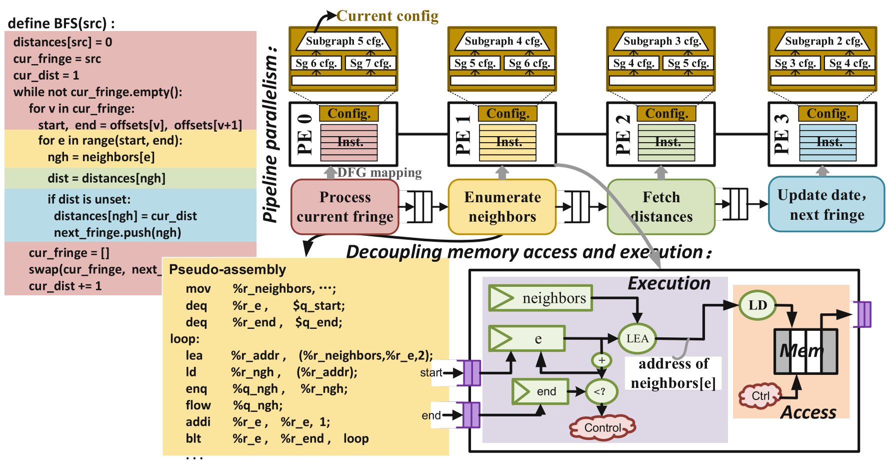


Fig. 2. (a) Percentage of irregular applications in several typical benchmark suites [11]. (b) Utilization of the fabrics using previous methods.

For a concrete example, we use BFS (Breadth First Search), a common graph algorithm that searches the distance from a source vertex *src* to all vertices reachable from it. BFS is a challenging irregular workload due to its multiple levels of indirection: it uses elements from cur_fringe to access offsets, which is then used to access neighbors, which in turn is used to access distances. It is a typical irregular application consisting of imperfect loop, complex control flow and unstructured memory access. Figure 3 shows the pseudo-code for BFS and illustrate its implementation on dataflow fabric using pipeline parallelism and decoupling access-execution [6].

Challenge 1: Load Imbalance. In Fig. 3 , the process current fringe node reads vertices from cur_fringe, whose neighbors are identified in the enumerate neighbors node. For each of these neighbors, the fetch distances node loads the distance



```
define BFS(src) :
  distances[src] = 0
  cur_fringe = src
  cur_dist = 1
  while not cur_fringe.empty():
    for v in cur_fringe:
      start, end = offsets[v], offsets[v+1]
      for e in range(start, end):
        ngh = neighbors[e]
        dist = distances[ngh]
        if dist is unset:
          distances[ngh] = cur_dist
          next_fringe.push(ngh)
    cur_fringe = []
    swap(cur_fringe, next_
    cur_dist += 1
```

Pseudo-assembly

```
    mov     %r_neighbors, ⋯;
    deq     %r_e ,     $q_start;
    deq     %r_end ,   $q_end;
loop:
    lea     %r_addr ,  (%r_neighbors,%r_e,2);
    ld      %r_ngh ,   (%r_addr);
    enq     %q_ngh ,   %r_ngh;
    flow    %q_ngh;
    addi    %r_e ,  %r_e, 1;
    blt     %r_e ,  %r_end ,  loop
    ...
```

Fig. 3. Illustration of breadth-first search (BFS) using pipeline parallelism and decoupling memory access and execution.

of this neighbor, which is checked against the current distance from the source by the update data node. By decomposing a large graph into multiple subgraphs, the BFS algorithm can be executed in a pipelined manner among these four PEs, as shown at the top of Fig. 3. After instruction assembly, the number and type of instructions are different between DFG nodes, and even the number of iterations is different. The enumerate neighbor node contains loop, address calculation. The updating node deals with branch, while the getting distance nodes only requires getting distance. The node with the longest delay among the four nodes will block the pipelined execution.

Challenge 2: Lack of Fine-Grained Pipelining Scheduling. The conventional, coupled load interface is a simple connection to the memory hierarchy and stalls the PE on cache misses. Simple memory access patterns, like streaming linearly through memory, do not need to be decoupled, and would be suitable for this interface, while some accesses are known to miss frequently, causing lengthy stalls. Decoupled architecture allows these accesses to be further from DFG execution, which is equipped with a small finite state machine within the PE, as shown in Fig. 3 (bottom). The access datapath now performs the memory access, which will obtain the neighbor id *ngh* as a result. Once this value is available, *ngh* is placed in the output queue to be sent to the consumer node. Even if a memory access to the neighbor array results in a cache miss, the enumeration neighbor node can still perform computations on other subgraphs at the same time, causing the DFG pipeline to stall only when the input queue of the computation is empty or the access queue is full.

However, for irregular applications, it is not enough to only decouple computation and memory access in a coarse-grained manner. In the dataflow-driven model, a DFG node can be fired only if its source operands are ready. Thus, for programs that have complex control flow and complex DFG structure, the data transfer (the *flow* operation in Fig. 3) needs to be executed as early as possible, because these data activate the consumer nodes. In addtion, these methods are limited to a program with small proportion of memory accesses, such as SHA (Secure Hash Algorithm) in Fig. 2.

4 Our Design

Our goal is to address the challenges described in Sect. 3. Figure 4 shows the process of transforming partitioned serial code into configurations for a dataflow fabric. We highlight our contributions using red lines, while other steps are common techniques in previous works [1,6]. We generate LLVM intermediate representation (IR) for each workload, and an automated tool examines the LLVM IR and produces a DFG using the actual operations that can be performed by PE's functional units.

In order to solve the load imbalance among DFG nodes, *DFG balancing* is introduced, which is a heuristic algorithm that achieves load balancing through instruction scheduling among DFG nodes. To exploit more parallelism and accelerate irregular applications, we propose *decoupled execution model*, a novel execution and schedule mechanism for DFG threads. Moreover, a *decoupled PE architecture* is provided to support the decoupled execution model efficiently.

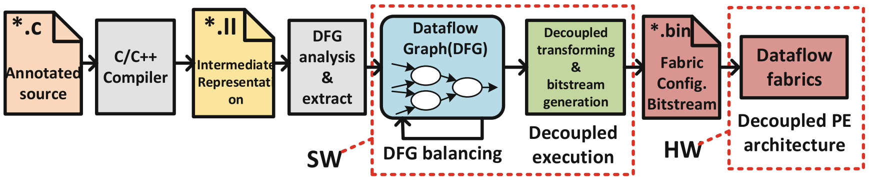


Fig. 4. Workflow of our methods.

4.1 Load Balancing

DFG balancing is a heuristic algorithm and it intends to dispatch instructions from high-load nodes to low-load nodes. Note that it is hard to generate an absolutely balanced DFG because: 1) the delay of each nodes is unpredictable during execution, like stalls caused by hazard or memory access. 2) allocating the same number of instructions to each DFG node is expensive and is limited by the applications itself, which often leads to non-convergence. Thus, we aim to generate a relatively balanced DFG based on the number and type of instructions.

The algorithm of DFG balancing is described in Algorithm 1. A DFG $G = (V, E)$ generated by the earlier stages in the toolchain (Fig. 4) and a threshold θ are the inputs. The first step is to sort the DFG nodes in depth-first order and estimate their latency (Line 1–4). When estimating the latency of each

Algorithm 1. Instruction Reschedule Algorithm

```
Input: a dataflow graph $G = (V, E)$, and a threshold $\theta \in \mathbb{Z}^+$
Output: a more balanced dataflow graph $G' = (V', E')$
 1: Init set $CP \leftarrow$ sortbyDFS($G$)                               ▷ Step ①
 2: Init $C_{num} \leftarrow$ getNumofNodes($CP$)
 3: Init $List[] \leftarrow$ getLatencyofEachNode($CP$,$inst\_latency$)
 4: Set $\chi \leftarrow \sum_{n=1}^{C_{num}}$ List[] / $C_{num}$
 5: for each node $n_i$ in $CP$ do                                         ▷ Step ②
 6:     Set $load \leftarrow$ List($n_i$)
 7:     if $load \geqslant \chi + \theta$ then
 8:         dispInst2Downstream($selInst$,$n_i$,$n_{i+1}$)
 9:     end if
10:     if $load \leqslant \chi$ - $\theta$ then
11:         dispInsfromDownstream($selInst$,$n_i$,$n_{i+1}$)
12:     end if
13: end for
14: return generate $G' \leftarrow$ refresh($G$, $CP$)                     ▷ Step ③
```

node (Line 3), we need to refer to the instruction type (*inst_latency*), because the execution time of different instructions may be different, which is related to the instruction set architecture (ISA). For simplicity, this evaluation only considers the number of instructions and their latency, and the PE only support partial RISC-V ISA (RV64I) and some dedicated dataflow instructions (*flow*). A comparison factor χ is used in Algorithm 1, which is calculated in Line 4, where the *List*[ ] array maintains the latency of each node on the critical path. It will be used as a reference in Step 2.

The principle of Step 2 is to find the imbalance DFG nodes and perform instruction redispatch (Line 5–13). The threshold θ and the comparison factor χ are used to obtain an interval (χ - θ, $\chi + \theta$). If a node's latency is in this interval, it is a suitable node. If a node's latency is greater (or less) than this interval's upper (or lower) bound, it can be seen a heavy (or light) node, respectively. For a heavy/light node, the algorithm will dispatch computing instructions to/from its downstream node. If a heavy node has no downstream nodes, the node will be split into two nodes. We found it difficult to find a threshold θ that fits all applications. The smaller the θ is, the more balanced DFG is generated, but Algorithm 1 becomes more complex and harder to converge. When the θ is larger, the overhead of Algorithm 1 will decrease, but the performance of the application will also decrease. We found that a good trade-off between performance and cost can be achieved when the θ is set in a range of [3,5]. The final step of Algorithm 1 is to update the DFG according to the adjusted CP and to generate the final DFG G'.

4.2 Decoupled Model

The decoupled execution model defines a novel scheme to schedule and trigger DFG nodes and exploit instruction block level parallelism. The code of each node consists of up to four consecutive stages: *Load stage, Calculating stage, Flow stage* and *Store stage*, which we describe below:

- Ld (Load) Stage: This stage loads data from the memory hierarchy to the in-PE local memory.
- Cal (Calculating) Stage: This stage completes calculations. A node can enter the Cal stage only when the following two conditions are met: first, its Ld stage (if it exists) has already finished; second, it has received all the necessary data from its predecessor nodes.
- Flow Stage: This stage transfers data from the current node to its successors.
- ST (Store) Stage: This stage transfers data from the in-PE operand memory to the memory hierarchy.

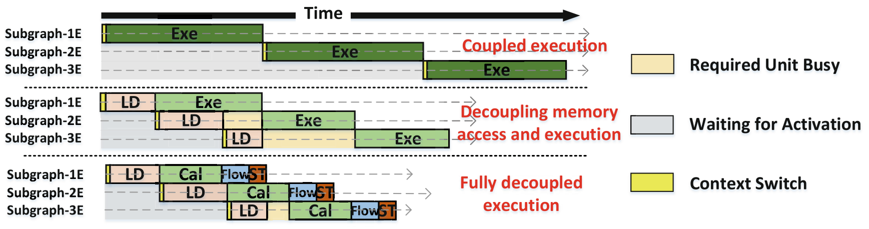


Fig. 5. Comparison of three different execution models.

Correspondingly, instructions in a DFG node will be rearranged according to their types and divided into four different blocks. The block is a basic schedule and trigger unit. Different from traditional out-of-order execution, the decoupled execution model exploits more instruction-block level parallelism without complex control logic, such as reorder buffer. Figure 5 takes the process of enumerating neighbor nodes of BFS (in Fig. 3) as an example to show the comparison between the decoupled execution model and the previous two execution models. In the coupled model (top), the execution of DFG nodes adopts a non-preemptive mechanism. The subgraph-1 will not release the PE resources until the end of execution. After decoupling the memory access in DFG node (middle), the subgraph-2 can perform the memory access operation after the LD stage of the subgraph-1 is finished. In this way, the PE can process up to two subgraphs at the same time. But the execution of subgraph-3 requires a long waiting delay. This is because the subgraph-2 occupies PE resources due to the coarse-grained (partial) decoupling. Fortunately, this problem can be addressed in the fully decoupled execution model (bottom). Through a more fine-grained scheduling mechanism, PE can process more subgraphs at the same time, and can overlap more delays, such as memory access and data flow.

4.3 Decoupled Architecture

Figure 6 illustrates the top-level diagram of our dataflow architecture, which is comprised of a set of identical decoupled processing elements (dPE). To support the decoupled execution model, separated four-stage components are designed within each PE to correspond to the four different states of the nodes. The function of the controller is to maintain and schedule the execution of different node states. And to ensure the correctness of the execution, separate operand RAM space is provided for different iterations. And a shared operand RAM space is set up to store the data that has dependencies between iterations, which are marked by special registers in the instructions.

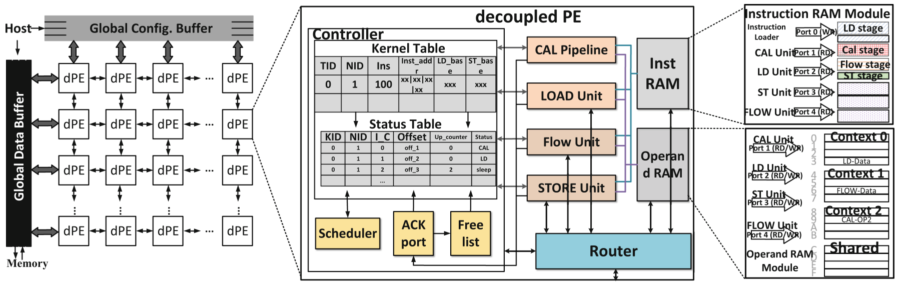


Fig. 6. The decoupled PE architecture.

The dPE consists of a calculation pipeline, a load unit, a store unit, a flow unit, an instruction RAM module, an operand RAM module, a controller and a router (in the middle of Fig. 6). These four separate functional components (CAL, LOAD, FLOW, STORE) and the controller are designed for the decoupled model, which are different from previous structures. The calculation pipeline is a data path for arithmetic operations and logical operations. It fetches instructions from the instruction RAM module and performs computations on source data. The load/store unit transfers data from/to on-chip data memory to/from operand RAM module, respectively. And the flow unit dispatches data to downstream dPEs. Each execution unit has a corresponding state, as described in Fig. 5, and such a decoupling method is the key to improving the utilization.

The controller plays a non-negligible role in the state transition and DFG nodes triggering. It consists of a kernel table, a status table, a free list, a dedicated acknowledgment buffer (Ack port), and a scheduler module. The kernel table stores the configurations of the nodes mapped to the dPE, which contain the task ID (*TID*), node ID (*NID*), instance number (*instance*), instruction address list (*inst_addr*) and data address (*LD_base&ST_base*). The *TID* and *NID* are used to identify task and DFG node, because the PE array can be mapped to multiple tasks at the same time, and a PE can be mapped to multiple nodes. The *instance* is a value related to the pipeline parallelism, which indicates how many times the DFG node needs to be executed. Taking BFS as an example, for a large graph, it may need to be decomposed into many subgraphs, such as 100, then each DFG node needs to be executed 100 times. The *inst_ addr* records the location of the four-stage instruction of the DFG node in the instruction RAM. The *LD_ base&ST_ base* are the base addresses for the source and destination, which can work with the *offset* in the status table to access the data in the operand RAM.

The status table maintains the runtime information for different instances. It uses the *instance_ counter* to record different instances of DFG nodes. Although different instances share the same instructions, they handle different data. Therefore, the offsets (*offset*) of different instances are different. In addtion, the status table records the activations (*Up_ counter*) and status informations. The value of *Up_ counter* decreases with the arrival of activation data. When this value is 0, it means that all the upstream data of the current node has arrived and it can be triggered by the scheduler module.

The scheduler uses the *instance_ counter* to evaluate the priority, and schedules nodes according to their priority. We also tried other scheduler policies, such as a round-robin scheduler or finer-grain multithreading, but found that these did not work as well. This makes sense: the application work done is nearly constant regardless of the scheduling strategy, so a simple scheduling mechanism is effective. Also, simple scheduling principles reduce configuration overhead. The Ack port is connected to the four pipeline units in order to obtain the status of each section. Additionally, the Ack port uses this information to dynamically modify the contents of the state table for scheduling by the scheduler. And the free list queue maintains free entries in this buffer.

The instruction RAM module consists of multiple single-port SRAM banks. Each bank can be occupied by a single functional unit at any time. The operand RAM module consists of multiple 1-write-1-read SRAM banks. To ensure the pipeline execution between instances, a separate context is allocated for each iteration. Considering that there may be dependent data between instances, a shared context is established in the operand RAM. Shared data are marked by special registers in the instructions.

5 Methodology

Setup. We develop a cycle-accurate micro-architecture simulator for hardware utilization and performance evaluation. The simulator is developed in C language based on SimICT framework [16] and can simulate behaviors such as memory access, data transfer, scheduling, etc. We calibrate the error to within ±7% using RTL environment. We also implement our architecture using Verilog. We use Synopsys Design Compiler and a TSMC 28nm GP standard VT library to synthesize it and obtain area, delay and energy consumption, which meets timing at 1 GHz. Table 1 shows the hardware parameters.

Table 1. Hardware Parameters.

Component		Parameter	Area (mm^2)	Power (mW)
dPE	Func. Units	INT&FP32	0.046(26.90%)	7.92(29.61%)
	Controller	–	0.012(7.27%)	1.20(4.97%)
	Inst. RAM	4KB	0.003(1.81%)	0.38(1.56%)
	Oper. RAM	64KB	0.812(47.27%)	9.93(41.18%)
	Routers	–	0.028(16.72%)	4.67(19.41%)
	Total		*0.1719*	*24.1019*
PE Array		8 × 8	11(79.38%)	1542(84.45%)
NoC		2D mesh	0.65(4.72%)	70.65(3.86%)
Glo. Data Buf.		1MB(SPM)	1.67(12.06%)	150.57(8.79%)
Glo. Conf. Buf.		0.2MB(SPM)	0.35(2.50%)	38.11(2.08%)
DMA		2 channels	0.19(1.37%)	14.65(0.8%)
Total			***13.8647***	***1826***

Benchmarks. To evaluate our methods, we use the benchmarks from Fifer [6] and literature [11]. These irregular workloads contain imperfect loops, complex control flow and unstructured access. And we used the same input data as those in the literatures [6,11]. Table 2 lists the selected workloads.

Table 2. Workloads for Evaluation.

Workload	Characteristic	Benchmark suite
GEMM, Viterbi(VIT) Sort, FFT	Imperfect loop Complex control flow	MachSuite adopt from [11]
CFD HotSpot(HS) LUD, GE	Imperfect loop Complex control flow Loop dependency	Rodinia adopt from [11]
Gesummv(GES) Cholesky	Imperfect loop Complex control flow	PolyBench adopt from [11]
BFS,PageRank CC, Radii	Unstructured access Imperfect loop	Fifer [6]

6 Evaluation

6.1 Results and Analysis

To evaluate the effectiveness of the methods we proposed, we implement the following four different experiments.

- **Baseline (Base).** It is our baseline, using only pipeline parallelism to accelerate irregular applications.

– **Baseline + DFG Reorganization (D1).** It combines the pipeline parallelism with DFG balancing technique.
– **Baseline + Decoupled Model & Architecture (D2).** It combines the pipeline parallelism with decoupled model and hardware.
– **Baseline + DFG Reorganization + Decoupled Model & Architecture (D3).** It combines the pipeline parallelism with our three methods.

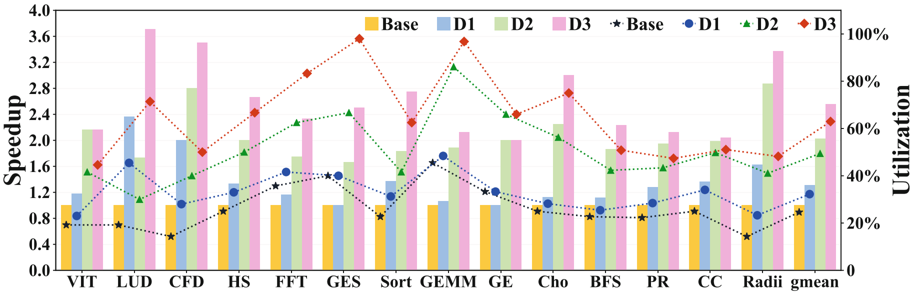


Fig. 7. Utilization (in marker) and speedup (in bar) over the baseline.

Figure 7 demonstrates the effectiveness of our proposed methods in terms of performance and utilization improvements. DFG balancing (D1) achieves an average performance improvement of 1.31×. Decoupled execution technique (D2) improves performance by gmean 2.03× over the baseline. By combining these approaches (D3), the performance of the dataflow fabric can be improved by 2.55×, and the average computing resource utilization has also reached 65.12%. In most cases, decoupled execution achieves better performance and utilization improvements compared to DFG balancing.

For imperfect loop like GEMM, Gesummv and GE, the inner and outer loops are almost equal in size and the load of each DFG node is more balanced. Thus the effect of DFG balancing is not very obvious while the improvement of the decoupled execution is obvious. Because decoupled execution can overlap the delays caused by memory access and data transfer and improve the utilization up to 96.8%. For dependency loop like LUD, data dependence reduces the utilization by limiting inter sibling loops parallelism and explicit data barrier also exacerbates the problem, which limit the effectiveness of decoupled model. For kernels with branches such as Sort, FFT and HotSpot, the utilization is significantly degraded in baseline, especially in Sort (only 22.7%), which has plenty of elseif statements. Our design decouples the data transfer stage so that activations can be delivered to downstream nodes as early as possible. Even though we didn't use prediction techniques, it still achieves a speedup of 2.75×.

Cost. The hardware overhead of decoupled execution is shown in Table 1. The area and power consumption of the controller used for scheduling in dPE only account for 7.27% and 4.97%, respectively. We evaluate Algorithm 1 on Intel(R)

Core(TM) i7-7700 CPU@2.80GHz. This time is 5.1% of the execution time on average, so it has negligible effect when performed at runtime.

6.2 Comparison with Other Dataflow Architectures

For comparison, we use three typical dataflow architectures, i.e. Plasticine [4], Fifer [6] and Yin et al. [11]. Plasticine features pipeline parallelism. Fifer features decoupling access and execution. Yin et al. [11] features subgraph rescheduling (detailed in Sect. 2). The hardware parameters of the three architectures are shown in Table 3, where we align them with similar peak performance. To model their performance and utilization, we leverage the open source implementations for Plasticine [4] and Fifer [6]. For work [11], we obtained data from the paper.

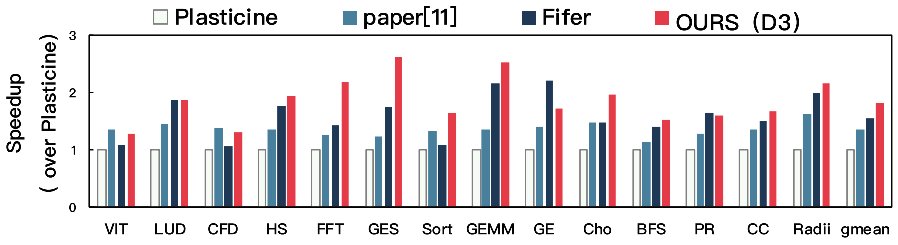


Fig. 8. Performance comparisons normalized to Plasticine.

Performance. Figure 8 illustrates the speedup comparisons normalized to Plasticine. Our design (D3) outperforms the Plasticine by gmean 1.81× and by up to 2.53×. This speedup comes from D3's ability to further shorten the interval between different iterations of the DFG pipeline execution. Compared with work [11], D3 achieves average 1.34× performance improvement. The reason for limiting the performance of paper [11] is that the execution of DFG nodes still adopts a coarse-grained mechanism, resulting in an average utilization of only 39.04%. Fifer achieves an average 1.54× performance improvement. These performance

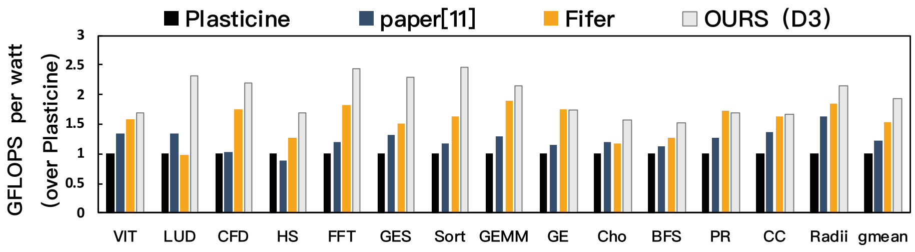


Fig. 9. Energy efficiency comparisons normalized to Plasticine.

gains come from decoupling memory access. However, for computationally intensive applications like VIT (1.09×) and CFD (1.05×), the improvement is not obvious.

Energy Efficiency. Figure 9 shows the energy efficiency (***performance-per-watt***) comparisons normalized to Plasticine. On average, Our design (D3) achieves 1.94× efficiency improvement over Plasticine, 1.58× over work [11] and 1.26× over Fifer. The coarse-grained scheduling mechanism employed in Plasticine results in lower utilization, resulting in poor energy efficiency performance. Work [11] achieves a relatively high energy efficiency in most cases by rescheduling at software level. But for HotSpot and CFD, it consumes more energy on buffer accesses due to the frequently subgraph switching. In Fifer, a large number of buffers are added between PEs to reduce the impact of load imbalance. But the energy overhead of these buffers is very large.

Table 3. Hardware Comparisons.

Arch	Plasticine [4]	Yin et al. [11]	Fifer [6]	**OURS(D3)**
Tech (nm)	28	28	28	28
Area (mm^2)	12.6	13.95	21.44	13.86
Power (W)	2.002	2.415	2.476	1.826
Freq (GHz)	1	0.8	2	1
PeakPerf (GFLOPS)	523	576	640	512
Efficiency (GFLOPS/W)	58.25∼99.79	27.85∼137.29	113.88∼218.17	116.64∼280.39

7 Conclusion

This paper presents a software and hardware co-design technique that makes both regular and irregular applications efficient on dataflow architectures. We propose an instruction schedule method to solve load imbalances and a more fine-grained scheduling and trigger mechanism. Experiments exhibited by our methods achieve significant utilization and performance improvement on key application domains with small modifications.

Acknowledgment. This work was supported by the National Key Research and Development Program (Grant No. 2022YFB4501404), CAS Project for Youth Innovation Promotion Association, Open Research Projects of Zhejiang Lab (Grant NO. 2022PB0AB01), Beijing Nova Program (Grant No. 2022079).

References

1. Wu, X., Fan, Z., Liu, T.: LRP: predictive output activation based on SVD approach for CNN s acceleration. In: DATE, pp. 831–836 (2022)

2. Ye, X., Tan, X., Wu, M., et al.: An efficient dataflow accelerator for scientific applications. Future Gener. Comput. Syst. **112**, 580–588 (2020)
3. Zhang, Y., Zhang, N., Zhao, T.: Sara: scaling a reconfigurable dataflow accelerator. In: ISCA, pp. 1041–1054 (2021)
4. Prabhakar, R., Zhang, Y.: Plasticine: a reconfigurable architecture for parallel patterns. In: ISCA, pp. 389–402 (2017)
5. Nguyen, Q.M., Sanchez, D.: Pipette: improving core utilization on irregular applications through intra-core pipeline parallelism. In: MICRO, pp. 596–608 (2020)
6. Nguyen, Q.M., Sanchez, D.: Fifer: practical acceleration of irregular applications on reconfigurable architectures. In: MICRO, pp. 1064–1077 (2021)
7. Ham, T.J., Aragón, J.L., Martonosi, M.: DeSC: decoupled supply-compute communication management for heterogeneous architectures. In: MICRO, pp. 191–203 (2015)
8. Weng, J., Liu, S., et al.: A hybrid systolic-dataflow architecture for inductive matrix algorithms. In: HPCA, pp. 703–716 (2020)
9. Taylor, M.B., Kim, J., et al.: The raw microprocessor: a computational fabric for software circuits and general-purpose programs. IEEE Micro **22**(2), 25–35 (2002)
10. Dadu, V., Nowatzki, T.: Taskstream: accelerating task-parallel workloads by recovering program structure. In: ASPLOS, pp. 1–13 (2022)
11. Yin, C., Wang, Q.: Subgraph decoupling and rescheduling for increased utilization in CGRA architecture. In: DATE, pp. 1394–1399 (2021)
12. Capalija, D., Abdelrahman, T.S.: A high-performance overlay architecture for pipelined execution of data flow graphs. In: 2013 23rd International Conference on Field programmable Logic and Applications, pp. 1–8 (2013)
13. Zhao, Z., Sheng, W., Jing, N., He, W., et al.: Resource-saving compile flow for coarse-grained reconfigurable architectures. In: ReConFig, pp. 1–8 (2015)
14. Kasgen, P.S., Weinhardt, M., Hochberger, C.: Dynamic scheduling of pipelined functional units in coarse-grained reconfigurable array elements. In: Schoeberl, M., Hochberger, C., Uhrig, S., Brehm, J., Pionteck, T. (eds.) ARCS 2019. Lecture Notes in Computer Science, vol. 11479, pp. 156–167. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-18656-2_12
15. Gobieski, G.: Manic: a vector-dataflow architecture for ultra-low-power embedded systems. In: MICRO (2019)
16. Ye, X., Fan, D., Sun, N., Tang, S., Zhang, M., Zhang, H.: SimICT: a fast and flexible framework for performance and power evaluation of large-scale architecture. In: ISLPED, pp. 273–278 (2013)