Minimizing Pipeline Stalls in Distributed-Controlled Coarse-Grained Reconfigurable Arrays with Triggered Instruction Issue and Execution

Yanan Lu, Leibo Liu*, Yangdong Deng, Jian Weng, Zhaoshi Li,
Chenchen Deng, and Shaojun Wei
Tsinghua National Laboratory for Information Science and Technology
Institute of Microelectronics, Tsinghua University, Beijing 100084, China
*Corresponding author: liulb@tsinghua.edu.cn

ABSTRACT
The pipeline stall in distributed-controlled coarse-grained reconfigurable arrays is a major source stumbling performance. This work presents a Triggered-Issue and Triggered-Execution (TITE) paradigm motivated from the Triggered Instruction Architecture (TIA) which converts control and data dependencies into predicate dependencies as triggers for spatial parallelism. TITE separately triggers the issuing and execution of instructions to further relax the predicate dependencies in TIA. Triggered dual instructions and tag forwarding are proposed to minimize pipeline stalls of both intra and inter-processing elements. Experiments show that TITE improves performance, energy efficiency, and area efficiency by 21%, 17%, and 12%, respectively, compared with TIA.

Categories and Subject Descriptors
C.1.3 [**Processor Architecture**]: Other Architectural Styles

General Terms
Performance, Design

Keywords
Coarse-grained reconfigurable array, Distributed-control, Pipeline stall, Triggered-issue and triggered-execution

1. INTRODUCTION
Reconfigurable computing architectures have received wide attention in both academia [1] and industry [2] due to their ability to deliver an optimal tradeoff between performance and programmability. Recent years have witnessed an ever-growing momentum to exploit reconfigurable hardware in mainstream computing infrastructures [3]. Among the various reconfigurable architectures, distributed-controlled Coarse-Grained Reconfigurable Arrays (CGRAs) are dynamically reconfigurable fabrics that can efficiently handle irregular control flows inside autonomous Processing Elements (PEs), thereby avoiding the heavy communication overheads between PEs and the main controller. Each autonomous PE is equipped with a controller and a pipelined data path, while multiple PEs in the array can be cascaded to form long pipelines via the interconnection. Since the pipelined data paths are the major computing resources of CGRAs, the minimization of pipeline stalls is essential to sustain the computing throughput [4]. In addition, distributed-controlled CGRAs, in which the PEs are autonomous and tightly coupled, employ pipelines of both intra and inter-PEs, posing unique constraints on the design and optimization of the array. The stalls in the pipeline of intra-PEs, which are caused by control and data dependencies, are like those in General Purpose Processors (GPPs). However, the stalls in the inter-PE pipelines, which are caused by waiting for items (data or control) transmitted from adjacent PEs, are different from GPPs. Surprisingly, few research efforts have been dedicated to the pipeline design space of CGRAs. To the best of authors' knowledge, this work is the first systematic study on the pipeline behavior of distributed-controlled CGRAs.

The conventional techniques for pipeline stalls, such as dynamic scheduling and branch prediction [5], are generally complicated and area-consuming. Moreover, techniques designed for CGRAs, such as partial predicate [6], full predicate [6, 7], parallel condition [8], and dual-issue single-execution (DISE) [9], mainly deal with stalls in centralized-controlled CGRAs, while having shortcomings or limitations for the stalls in distributed-controlled CGRAs. Among the previous techniques for CGRAs, the Triggered Instruction Architecture (TIA) [10, 11], which employs a distributed-controlled framework with dataflow-driven PEs, achieves the best performance for control flows [12]. TIA utilizes predicates instead of the Program Counter (PC) to allow programs to transit concisely between states without explicit branch instructions. Both control and data dependencies in TIA are converted into predicate dependencies. However, pipeline stalls remain because of the dependency between the issuing of a predicated instruction and the execution of an instruction that generates the predicate.

This paper presents an improved control paradigm, named Triggered-Issue and Triggered-Execution (TITE), that minimizes pipeline stalls in distributed-controlled CGRAs. Although TITE is motivated by TIA, a major difference is that the issuing and execution of instructions are separately triggered to relax the predicate dependencies. Firstly, when dealing with the pipeline stalls of intra-PEs, a dual instruction (i.e., a pair of two sequential instructions or instructions from each of the two branches) is triggered to issue without (waiting for the completion of the former instruction) and then remains pending. Once the former instruction is

*This work was supported in part by the National High Technology Research and Development Program of China (Grant No. 2012AA012701) and the National Natural Science Foundation of China (Grant No. 61672317).

DAC '17, June 18-22, 2017, Austin, TX, USA
 ISBN 978-1-4503-4927-7/17/06$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062284

finished, the qualified half of the dual instruction is triggered to execute. Thus, the issue-time of the predicated instructions is concealed and the branch stalls are eliminated. Experimental results show that execution time can be averagely reduced by 12%. Secondly, for stalls in the pipeline of inter-PEs, the timing relationship between the transmitted data and their tags are relaxed, while their order is maintained. Therefore, the tags can be forwarded for triggering a subsequent instruction to issue before the arrival of the corresponding data. The instruction that uses the data as an operand are issued and remain pending until the data is available. Thus, the pipeline stalls of inter-PEs caused by instruction switching are reduced and the execution time of applications is reduced by 9% on average. Experimental results show that although TITE has a greater area cost (8% overhead) than TIA, the performance, energy efficiency (i.e., performance/power), and area efficiency (i.e., performance/area) are all enhanced.

2. MOTIVATION

Generally, data and control flows in each application are tightly coupled. In a centralized-controlled CGRA that consists of a controller and an array of non-autonomous PEs, such as DySER [14], control flows are handled in the controller, which causes frequent interactions between PEs and the controller. Such interactions cause large communication overheads and significantly penalize performance [8]. A distributed-controlled CGRA, such as TIA [10], alleviates this problem by handling control flows inside the PEs because each PE is autonomous and can make localized decisions. However, such flexibilities bring extra stalls as well. Since multiple instructions are allocated to autonomous PEs, a PE should choose to issue an instruction before execution. Unlike the case in conventional processors, the issuing of instructions in an autonomous PE is dependent on the intra-PE states and the messages or data from the adjacent PEs. As a result, a data dependency in the pipeline of inter-PEs becomes a control dependency. A PE must stall the pipeline before the arrival of the required items from its neighbors. Pipeline stalls occur because the issue of the subsequent instruction is not concealed. Moreover, in the intra-PE pipelines, both branch and data hazards can cause stalls, which is the same as the case in conventional processors. These common problems have not yet been adequately resolved in distributed-controlled CGRAs.

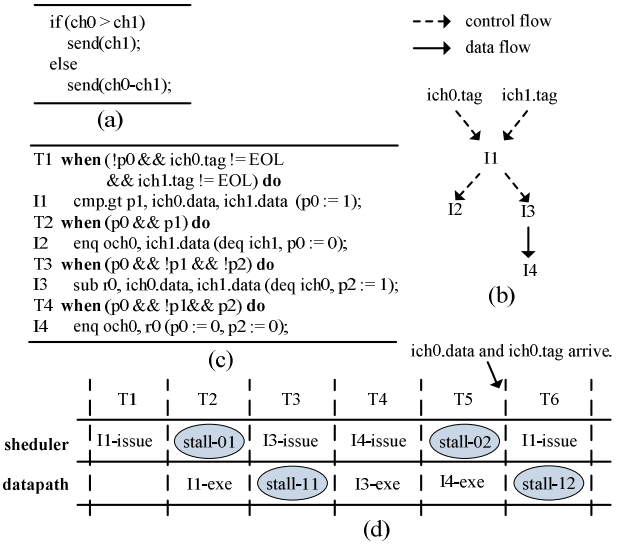

Figure 1. Illustration of impact of pipeline stalls. (a) Source code; (b) CDFG; (c) Triggered instructions and triggers; (d) Execution Pipeline.

To illustrate the impact of pipeline stalls in CGRAs, Figure 1 shows an example of a piece of code executed on TIA which is the state-of-the-art [10, 12]. The code compares the data from two input channels that buffer data transmitted from adjacent PEs and sends results to the output channel for buffering and transmission, as shown in Figure 1(a). Figure 1(b) shows the Control and Data Flow Graph (CDFG). Figure 1(c) shows the triggered instructions and their triggers according to the TIA specifications. Figure 1(d) illustrates the execution pipeline of the code, assuming that *I2-I4* is the taken branch. The executions are assumed to complete in one clock cycle since the instructions are simple operations. As shown in Figure 1(d), there is a stall at the *T2* cycle caused by control dependency and a stall at *T5* cycle caused by data dependency on an adjacent PE. Together, they comprise **40%** of the total execution time. If the stalls are eliminated, then the performance is significantly enhanced.

There have been some conventional techniques for pipeline stalls, such as dynamic scheduling and branch prediction [5]. Although these techniques can also be effective for pipeline stalls in autonomous PEs, they are generally complicated and require heavy overheads on area, which contradicts the principle of CGRAs that aim for high energy and area efficiencies. TIA, which utilizes triggered instructions instead, can achieve 8× greater area-normalized performance than a conventional GPP [10]. Previous methods proposed for stalls in CGRAs mostly focused on control dependencies in a centralized-controlled fabric. However, these methods have severe limitations on handling stalls in distributed-controlled CGRAs. Partial predicate [6], control-based full predication [6] and state-based full predication [7] exploit instruction level parallelism by converting control dependencies into data dependencies, and accelerate control flows by utilizing abundant PEs to execute the predicated instructions in parallel. Although the parallelism is enhanced, the execution of unnecessary instructions occupies many available resources and the performance is decreased when the incorrect branch is longer. Parallel condition [8] and TLIA [13] also accelerate control divergence by parallel execution, but they make specific demands on the mapping of instructions and the data path of PEs, respectively, and pipeline stalls still remain after the divergence. DISE [9] pairs instructions from different branches for control divergence, but the code size is greatly enlarged (70% in [9]) because many *NOP* instructions are added to pair with sequential codes and *unbalanced-if*. The abovementioned TIA employs a hardware scheduler instead of the data path to generate the address of the next instruction, thereby eliminating the program counter and branch instructions. It exceeds all the other techniques in performance of control flows [12]. But pipeline stalls still remain.

Pipeline stalls in distributed-controlled CGRAs are mainly caused by control and data hazards in the pipelines of both intra and inter-PEs. Since both control and data dependencies in TIA are converted into predicate dependencies, the pipeline stalls caused by both types of hazards can be minimized by relaxing the predicate dependencies. An improved paradigm named TITE is proposed and detailed in the following sections.

3. TITE CONTROL PARADIGM

3.1 The Framework for TITE

The TITE paradigm is implemented on a distributed-controlled CGRA based on TIA. It is a scalable framework that consists of distributed scratchpad slices and autonomous PEs that connect together via a 2D-mesh network, as shown in Figure 2. Each PE comprises an instruction pool, hardware scheduler, data path, and

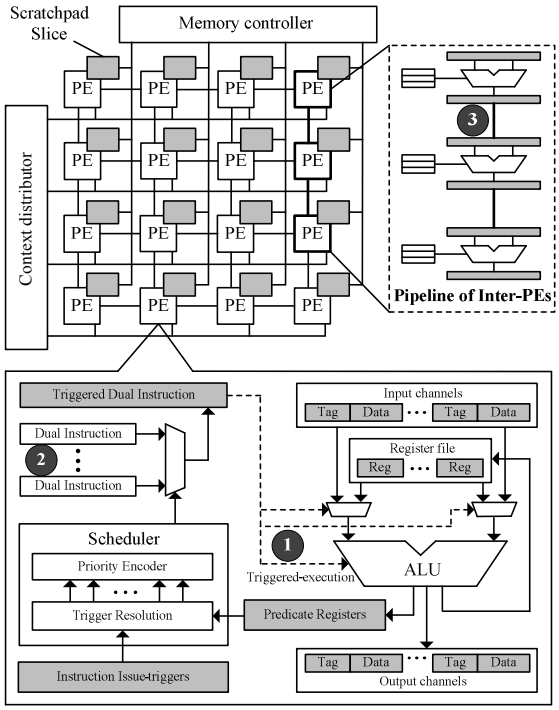

Figure 2. Framework for TITE paradigm.

input and output channels. Instructions are triggered by the scheduler and executed on the data path. A trigger (i.e., a Boolean expression built from a logical conjunction of a set of queries on the architectural state of the PE) is used to guard the instruction. Instructions whose triggers are met are issued for execution. The triggers are evaluated by the scheduler which comprises a trigger resolution and a priority encoder. The trigger resolution is a combinational logic circuit accepting channel status, tags, and predicate register values that are updated by the output of arithmetic logic unit (ALU) or programmer specification. The ALU finishes most instructions in one clock cycle, excepting the multiply and load instructions. Here, the floating point instructions which are orthogonal to the purpose of this paper are not implemented. Table 1 lists the architectural parameters, which are the same as those in [10].

Table 1. Architectural parameters

Sources per Instruction	2
Registers per PE	8
Predicates per PE	8
Max Triggered Instructions per PE	16
Network	Mesh(1 cycle link latency)
Scratchpad	8KB(distributed)

The pipeline technique in conventional processors is utilized in PE design. Pipelines of inter-PEs are formed when multiple PEs are cascaded via the interconnection. The execution flow proceeds as follows. Firstly, the context distributor dispatches a slice of instructions and their triggers to each PE. Secondly, the scheduler evaluates the issue-triggers of instructions in the instruction pool and selects the one with the highest priority from the candidates whose issue-triggers meet the issuing requirements. Finally, the triggered instruction is issued to the data path and starts to execute when its execution-trigger is met.

There are three major improvements in the TITE paradigm compared with TIA (as marked in black dots in Figure 2):

1) *The issuing and execution of instructions are separately triggered.*
2) *Instructions are paired to form dual instructions.*
3) *Tags bypass the output channel if the destination channel is not full.*

3.2 TITE for Pipeline Stalls of Intra-PEs

Pipeline stalls may happen because of control or data hazards between instructions of intra-PEs. Since data in a scratchpad slice can be fast accessed without conflicts and the data forwarding technique is employed in PEs to minimize stalls caused by data hazards, the control divergence is the main reason for pipeline stalls of intra-PEs. Unlike TIA, TITE can issue predicated instructions without stalls.

In TITE, where there is a control divergence, the first instruction from both branches is paired in a dual instruction at a fixed location. The two instructions share one issue-trigger that is independent from the predicate and a common execution-trigger that implies the arriving of the predicate. Thus, the two branch instructions can be issued when the predicate is under computation and they are pended until the release of the predicate. They oppositely react according to the predicate during execution. *Instruction_1* is the taken path when the predicate is 1 while *Instruction_0* is the taken path when the predicate is 0, as shown in Figure 3. The rest instructions in both branches are treated as sequential codes unless there is an inner control divergence. Each two consecutive instructions in sequence are arranged in a pair and a single sequential instruction is paired with *NOP*. This is different from DISE [9], which pairs every sequential instruction with *NOP* as well as the unpaired instructions in *unbalanced-if*. The *switch-case* structures that have more than two branches are converted to nested *if-else* structures.

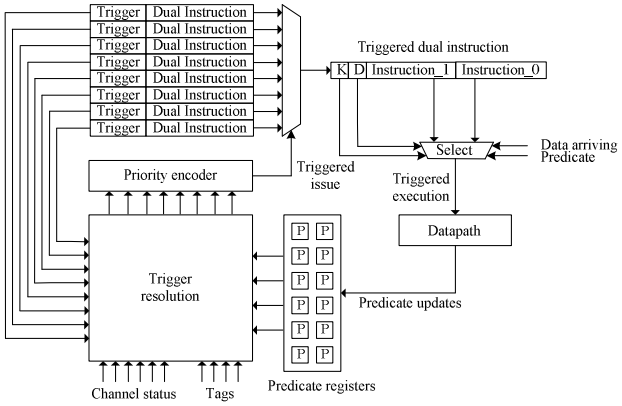

Figure 3. Microarchitecture of hardware scheduler.

A 2-bit tag is added to each dual instruction, as shown in Figure 3. Table 2 lists the tag definitions. Here, *K* indicates whether the instructions are branches and *D* indicates whether the second half is *NOP*. The controller of the data path recognizes the tag and executes instructions accordingly.

Table 2. Tag meanings in a dual instruction

	***K* (*Kind*)**	***D* (*Dual*)**
0	sequential instructions	*Instruction_0* is *NOP* which can be skipped.
1	Predicated instructions from different branches	*Instruction_0* is not *NOP* and cannot be skipped.

Given that two instructions in a pair share the same issue-trigger, the total cost for issue-triggers is smaller than that of TIA. The execution-trigger for each instruction is just a few bits that indicate the arriving of the required operands and the value of a predicate; as such, the hardware overhead cost is small.

The triggered dual instructions of the source code in Figure 1(a) are shown in Figure 4(a). Figure 4(b) illustrates the execution pipeline on TITE. It can be observed that the stall of *stall-01* in Figure 1(d) is eliminated by relaxing the dependency between the issuing of *I3* and the execution of *I1*. In this example, the execution time is reduced by 20%. To generate dual instructions, a modified compiler is required and is presented in section 3.4.

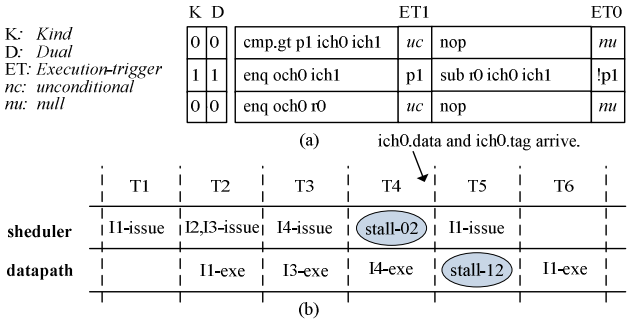

Figure 4. Example of triggered dual instructions and their execution. (a) Triggered dual instructions; (b) Execution pipeline.

3.3 TITE for Pipeline Stalls of Inter-PEs

Like TIA, PEs in the framework are interconnected by latency-insensitive channels which are implemented via static virtual circuits [11]. The control and data flows are mapped onto the fabric by the mapper (or programmer) on a hop-by-hop basis, while maintaining nonblocking and deadlock freedom via reverse credit flows. A PE can also be programmed to operate as a network router in a multi-hop traversal. The latency-insensitive channels use both input and output buffers for transmitted data.

The buffer in each channel is implemented as First-In-First-Out (FIFO) to allow multiple data as well as transmitting data in order. When an operand is in transmission, a programmer-specified tag is piggybacked onto it to identify the data. Control flows of inter-PEs are also translated into tags. The arrival or absence of a message (data or tag) is an implicit form of control synchronization. The tags (i.e., the *NotEmpty* signals of input channels and the *NotFull* signals of output channels) are sent to the scheduler as predicates for triggering a subsequent instruction.

Since tags are programmer-specified without computation, a tag can be sent to the output buffer when the corresponding data is under computation. If the destination channel is not full, it implies that the output channel is empty. Thus, the tag can bypass the output channel and be directly forwarded to the destination channel, without disturbing the order, as shown in Figure 5. The tag is used to trigger an instruction to issue and then the instruction is triggered to execute as soon as the corresponding data is available. A 1-bit valid signal is needed for the tag and data, respectively. However, the tag cannot bypass the output channel if the destination channel is full.

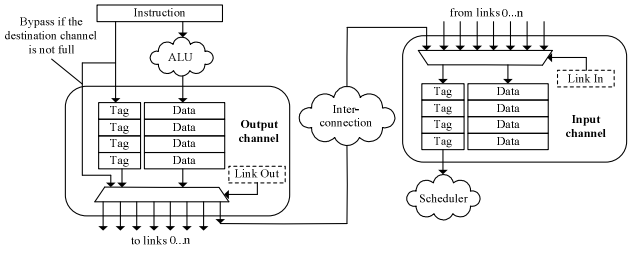

Figure 5. Illustration of tag bypassing the output channel.

Relaxing the timing between the data and tags can effectively reduce the stalls of inter-PEs caused by predicate dependencies. Figure 6 shows the execution pipeline after improvement. It can be observed that the stall of *stall-02* in Figure 1(d) is also eliminated and another 20% of the execution time is saved. A minor modification to the hardware is needed, however, it has negligible overhead costs on the area and does not affect the compiler.

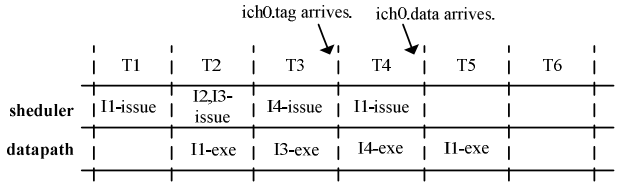

Figure 6. Execution pipeline after relaxing data and tag.

3.4 Compilation and Mapping Toolchain

The compilation and mapping toolchain for TITE is developed based on TRMap [12]. As TITE executes triggered dual instructions, corresponding modifications are made to the compilation. Figure 7 shows the compiling and mapping flow, where there are additional steps before and after the TRMap. Firstly, the applications source code is transformed into CDFG via Single Static Assignment (SSA) transformation. Secondly, all control divergence with a single branch in the CDFG are located and the default branches are filled with a *NOP* instruction, which results in a Modified CDFG (M-CDFG). Thirdly, the TRMap approach is used for generating triggered instructions and mapping the instructions onto PEs. Finally, the instructions for each PEs are paired to form the triggered dual instructions.

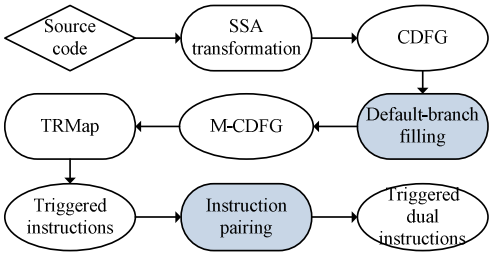

Figure 7. Compilation and mapping toolchain for TITE.

In the instruction pairing step, the triggered instructions generated by TRMap are paired following three rules. First, each two successive instructions in sequential codes are paired. The instruction of a predicate computation is treated as the last instruction of a sequence. If only a single instruction remains, then a *NOP* instruction is paired with it. Second, the first instruction from each of the two branches after a predicate computation are paired at fixed locations. Third, each branch without the first instruction is treated as a sequence and handled with the first rule. When two triggered instructions are paired for a dual instruction, the issue-trigger and execution-trigger are generated based on the kind of the pair. If it is a pair of predicated instructions, the trigger generated by TRMap is used as the shared issue-trigger, thereby remov-

ing the predicate. The predicate is then used as the execution-trigger. Otherwise, it is a pair of sequential instructions. The trigger of the first instruction is used as the shared issue-trigger; the execution-trigger of the second instruction is the completion of the first one while the first one is unconditionally executed.

4. EXPERIMENTAL EVALUATION

4.1 Setup

Given that only the relative performance of TIA has previously been reported and that the original performance is not available, the CGRAs based on TIA and TITE are implemented in Verilog Hardware Description Language (HDL) for comparison. In addition, improved architectures based on TIA for pipeline stalls of intra-PEs (TIA_AS) and inter-PEs (TIA_ES) are also realized and simulated to quantify the corresponding effect. For a fair comparison, each of the four architectures have the same microarchitecture, except for the abovementioned differences. The Register Transfer Level (RTL) implementations are synthesized in Synopsys Design Compiler with 65nm TSMC CMOS technology. The simulations are conducted with Synopsys VCS and the dynamic power is reported by Synopsys PrimeTime.

Table 3. Kernels for Evaluation

	Workload	Berkeley Dwarf	Domain
1	AES	Combinational Logic	Cryptography
2	Dense Matrix Multiply	Dense Linear Algebra	Scientific Computing
3	FFT	Spectral Methods	Signal Processing
4	Flow Classifier	Finite State Machine	Networking
5	BFS	Graph Traversal	Supercomputing
6	K-means	Dense Linear Algebra	Data mining
7	KMP String Search	Finite State Machine	Various
8	Merge Sort	Map Reduce	Database
9	SHA-256	Combinational Logic	Cryptography
10	Autcor	Structured Grids	Signal Processing
11	Prim	Graphical Models	Graph Theory
12	Rspeed01	Back-track and Branch + Bound	Speed Calculation
13	DPMM	Dynamic Programming	Scientific Computing
14	SpMV	Sparse Linear Algebra	Scientific Computing
15	Unstructured	Unstructured Grids	Graph Theory
16	Nbody	N-body Methods	Astrophysics

Application kernels from various domains are selected as benchmarks. In addition to the inclusion of the nine kernels used in [10], another seven kernels are also added for a full cover of the Berkeley Dwarfs [15], which abstract most of the known computing models and are widely accepted in academia [16]. Table 3 lists the selected kernels.

To simulate the kernels on the four architectures, the triggered instructions of the critical loops from the workloads are manually generated since there is no automatic tool. The instruction level parallelism is developed during mapping and the instruction mappings for each of the four architectures are the same. To avoid the impact of variable cache access latency, the data is loaded in the scratchpad memory before running the kernels. The data cache and context cache in all implementations are excluded during the synthesizing.

4.2 Performance and Energy Efficiency

Table 4 lists the clock cycles consumed by the different kernels that are executed on the four architectures. TIA_AS, TIA_ES and TITE averagely achieve 12%, 9% and 21% reduction in cycle numbers, respectively, compared with TIA. Considering the maximum working frequencies which are inversely proportional to the critical paths (as shown in Table. 5), ***TIA_AS, TIA_ES and TITE have performance increases of 10%，10% and 21%, respectively***, as shown in Figure 8.

Figure 8 shows that TITE has different effects on different kernels. For kernels that are control-intensive, such as Flow Classifier, BFS, and Merge Sort, TIA_AS has a significant performance improvement compared with TIA, while for kernels that are compute-intensive and spatially mapped on many PEs, such as DMM, Unstructured, and Nbody, TIA_ES significantly outperforms TIA. This is not surprising because the control-intensive kernels generally contain a large proportion of control divergence that can be sped up by the triggered dual instructions while the spatially mapped kernels gain benefits from the improved transmission between the PEs. As shown in Table. 5, the critical path of TIA_AS, TIA_ES, and TITE are longer than that of TIA. This is because the modifications to the microarchitecture have a penalty on the critical path. Thus, the maximum working frequencies of the three improved architectures are lower than that of TIA. However, they gain much greater reductions on execution cycles because of decreases in pipeline stalls and the overall performance is improved compared with TIA.

Table 5 also lists the average power consumption for running the kernels on the four implementations. TIA_AS, TIA_ES, and

Table 4. Execution cycles of kernels on the four architectures

#(cycles)	AES	DMM	FFT	Flow Classifier	BFS	K-means	KMP Search	Merge Sort	SHA-256	Autcor	Prim	Rspeed01	DPMM	SpMV	Unstructured	Nbody	*Normalized Mean.*
TIA	1186	384	3211	4256	357	89736	774	1920	3090	6819	829	2869	1319	12043	12551	17971	**1**
TIA_AS	1031	376	3058	3414	276	72044	661	1568	3008	6226	672	2362	1194	10711	12549	16954	**0.88**
TIA_ES	1111	336	2819	3838	353	88134	730	1824	2677	6755	797	2560	1235	10775	10247	14715	**0.91**
TITE	955	328	2666	2996	272	70442	617	1472	2595	6162	640	2053	1110	9443	10245	13698	**0.79**

Figure 8. Performance comparisons among four architectures normalized to TIA

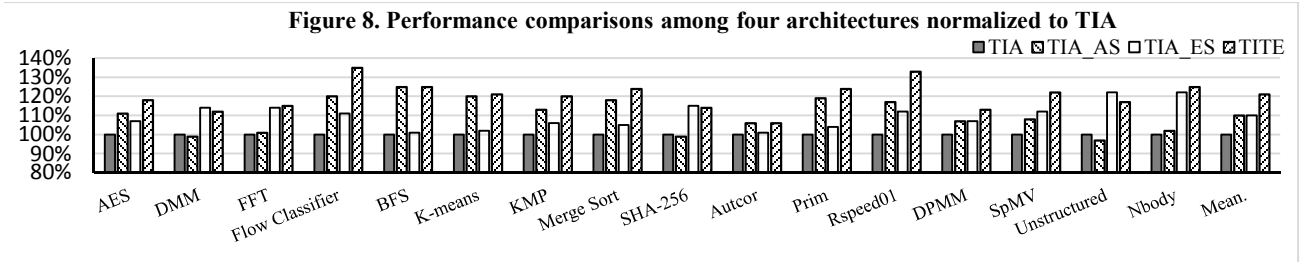

Table 5. Comparisons among the four architectures on ASIC

Architecture	TIA	TIA_AS	TIA_ES	TITE
Technology	TSMC 65nm 1P8M			
Number of PEs	16 (4 row × 4 column)			
Cell area(um^2)	1,383,612	1,460,658	1,409,309	1,491,379
Critical path (ns)	1.65	1.71	1.65	1.73
Dynamic Power (mW)	311	314	324	321

TITE have minor increases in dynamic power compared with TIA. The reason is that dynamic power is related to both area and clock frequency. TIA_AS costs more area than TIA, but runs at a lower frequency, which results in equivalent power consumption to TIA. TIA_ES runs as fast as TIA and TITE runs at a lower frequency, but they cost more area, leading to an increase in power consumption. However, since TIA_AS, TIA_ES, and TITE gain more increments on performance, ***their energy efficiencies (i.e., performance/power) are improved by 9%, 6%, and 17%, respectively, compared with that of TIA***.

4.3 Area and Area Efficiency

As listed in Table 5, ***the cell area of TIA_AS, TIA_ES, and TITE increase by 6%, 2% and 8%, respectively, compared with TIA.*** The reason for a notable increase on TIA_AS and TITE is the enlargement of the instruction pool in each PE. The pool is widened for buffering dual instructions and the capacity is increased by 20% for the added *NOP* instructions. Thus, the area of each pool increases by 39%. The area of TIA_ES slightly increases due to the additional logic for the triggered execution and tag forwarding. However, ***for area efficiency (i.e., performance /area), TIA_AS, TIA_ES, and TITE make improvements of 4%, 8%, and 12%, respectively, compared with TIA***. The four architectures are also implemented on FPGA. Table 6 lists the results which are consistent with that of the ASIC implementations.

Table 6. Comparisons among the four architectures on FPGA

Architecture	TIA	TIA_AS	TIA_ES	TITE
FPGA type	Xilinx xc7v2000tflg1925-2			
Slice LUTs	46,062	44,924	48,746	48,518
Slice Registers	19,104	19,657	29,317	29,846
Block RAM (36Kbit)	24	32	24	32
DSP module	192	192	192	192
Working frequency (MHz)	175	156	169	151
Dynamic Power (mW)	421	495	460	581

5. CONCLUSION

This paper presents a TITE paradigm that separately triggers the issuing and execution of instructions by utilizing triggered dual instructions and tag forwarding to relax the predicate dependencies for minimizing the pipeline stalls in distributed-controlled CGRAs. The paradigm effectively reduces the execution time and enhances the performance of CGRAs at the expense of minor overheads on area. Based on TIA, the proposed paradigm is implemented with modifications to the hardware structure and compiling approach. Evaluation on various application kernels verifies the effectiveness of TITE and shows enhancements on performance, energy efficiency, and area efficiency.

6. REFERENCES

[1] R. Tessier, K. Pocek, and A. DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3): 332-354, 2015.

[2] Intel's future with field programmable arrays: We're here to stay. https://newsroom.intel.com/press-kits/2016-idf/#2016-intel-developer-forum.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13-24. IEEE, 2014.

[4] J. Pager, R. Jeyapaul, and A. Shrivastava. A software scheme for multithreading on CGRAs. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(1):19, 2015.

[5] A. Seznec, S. Felix, V. Krishnan, et al. Design tradeoffs for the alpha EV8 conditional branch predictor. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 295-306. IEEE, 2002.

[6] S. A. Mahlke, R. E. Hank, J. E. McCormick, et al. A comparison of full and partial predicated execution support for ILP processors. In *Computer Architecture, 1995. Proceedings, 22nd Annual International Symposium on*, pages 138-149.

[7] K. Han, S. Park, and K. Choi. State-based full predication for low power coarse-grained reconfigurable architecture. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1367-1372. IEEE, 2012.

[8] J. Zhu, L. Liu, S. Yin, et al. A hybrid reconfigurable architecture and design methods aiming at control-intensive kernels. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9):1700-1709, 2015.

[9] K. Han, J. K. Paek, and K. Choi. Acceleration of control flow on CGRA using advanced predicated execution. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 429-432. IEEE, 2010.

[10] A. Parashar, M. Pellauer, M. Adler, et al. Triggered Instructions: a control paradigm for spatially-programmed architectures. In *Computer Architecture, 2013. Proceedings, 40th Annual International Symposium on*, pages 142-153, 2013.

[11] M. Pellauer, A. Parashar, M. Adler, et al. Efficient control and communication paradigms for coarse-grained spatial architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(3):10, 2015.

[12] S. Yin, P. Zhou, L. Liu, et al. Trigger-centric loop mapping on CGRAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1998-2002, 2016.

[13] L. Liu, J. Wang, J. Zhu, et al. TLIA: Efficient reconfigurable architecture for control-intensive kernels with triggered-long-instructions. *IEEE Transactions on Parallel & Distributed Systems*, 27(7):1-1, 2016.

[14] V. Govindaraju, C. H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 503-514. IEEE, 2011.

[15] K. Asanovic, R. Bodik, B. C. Catanzaro, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[16] K. Krommydas, W. Feng, C. D. Antonopoulos, et al. OpenDwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems*, 1-20. 2015.