A static-placement, dynamic-issue framework for CGRA loop accelerator

Zhongyuan Zhao*, Weiguang Sheng*, Weifeng He *, ZhiGang Mao* and Zhaoshi Li†
*Department of Micro/NaNo Electronics, Shanghai Jiao Tong University, Shanghai, China
†Institution of Micro Electronics, Tsinghua University, Beijing, China
*Email: zyzhao.sjtu@gmail.com

***Abstract*—This paper presents a static-placement, dynamic-issue (SPDI) framework for the coarse-grained reconfigurable architecture (CGRA) in order to tackle the inefficiencies of the static-issue, static-placement (SISP) CGRA. This framework includes the compiler that statically places the operations and hardware design, a SPDI CGRA, that automatically schedule the operations. We stress on introducing the SPDI CGRA in this paper. This newly designed hardware model adds the token buffer, which is capable of automatically scheduling the operations inside processing elements (PE), along with a router network that can effectively transform and control data flow among the PE array. This design lets the hardware share the responsibility for the compiler, making them cooperate to deal with the issuing, placement and routing problem. Evaluation of our study shows that our framework can reach on average 1.28, 1.30 and 1.33 higher than three state-of-the-art SISP CGRA using REGIMap, RS compile flow and the EPIMap approaches respectively. The area overhead is nearly 0.93% per token buffer entry for each PE relative to SISP CGRA.**

I. Introduction

Coarse-Grained Reconfigurable Architecture (CGRA) is one of the most attractive platforms for computationally intense applications considering both the performance and flexibility. It is often used for accelerating loops inside the kernel of the applications.

During the past decades, research works on CGRA have always been focus on the framework of the *static issue, static placement* (SISP) execution model (Fig.1). The performance of the SISP CGRA framework critically hinges upon the capabilities of the compiler. Since 2002, the compiler techniques based on the SISP CGRA have been developed, including the simulated annealing-based modulo scheduling [1], edge-centric modulo scheduling [2], graph embedding [3], graph minor [4], force-directed mapping [5] and sub-graph matching based approaches such as EPIMap [6], REGIMap [7] and RS compile flow [8]. These compilers have two common characteristics: (1) they statically issue and place the operations; (2) they use software pipelining based approach to optimize the loop execution and minimize the initiation interval (II) between two consecutive iterations. However, due to the SISP CGRA, the compilers face two serious performance challenges: (i) the CGRA mapping problem is

This work is supported by National High Technology Research and Development Program of China (863 Program), No.2012AA012702 and National Science Foundation of China under grant, No.61201059.

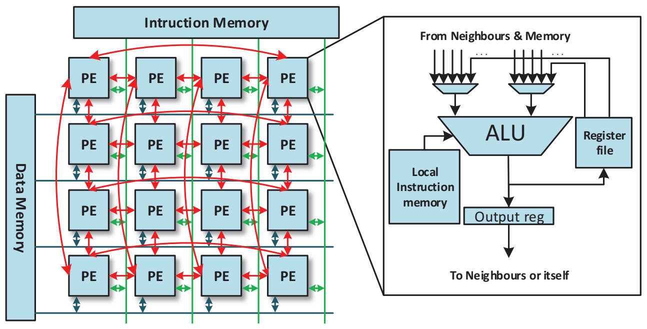
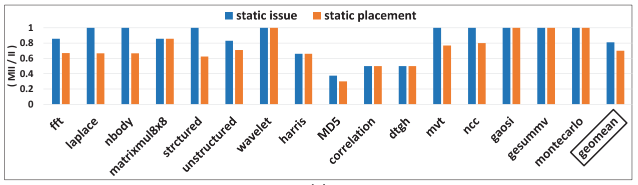


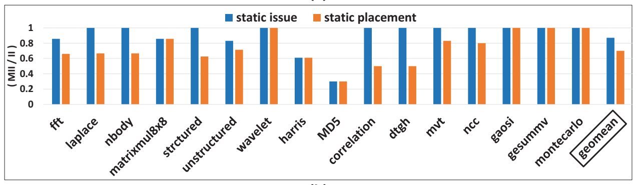
Fig. 1. The conventional CGRA computing model with torus topology [1]–[8], consists of an 4x4 processing elements (PEs) array, a shared data memory and an instruction memory. Each PE consists of an arithmetic logic unit (ALU),a local registers file, and a local instruction memory.

NP-complete, the heuristic based solution or the approximation algorithm can't guarantee the optimistic performance; (ii) they have to explicitly analyze the issue time for each operation without violating every mapping constraint, and effectively place and route them with least II - so many tasks pressed onto the compiler making it difficult to reach the optimized performance. The II may significantly increase during the compiling phase, which greatly reduces the performance. This pushes us to move forward to a new CGRA framework. In this framework, the hardware shares some responsibilities for the compiler. Therefore, they can work in cooperation to deal with the issuing, the placement and the routing problems, especially for the purpose of narrowing the gap between the optimal and the realistic performance.

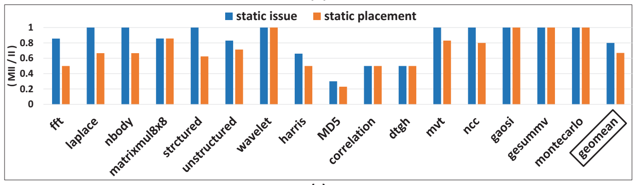
In this paper, we propose the *static placement, dynamic issue* (SPDI) CGRA. It is combined with a dynamic dataflow execution model to effectively accelerate the loop. The compiler statically maps operations in the data flow graph (DFG) onto the processing elements (PE) and configures the interconnect to transfer operands among PEs based on the graph's connectivity. The SPDI CGRA then dynamically schedules and routes the operations and automatically initiate the new iterations of the loop. When multiple operations are mapped onto the same PE, the *level* tag is labeled as priority to distinguish the operations. Whenever one operation wins the competition, it is issued to the ALU and then immediately sent to its consumer. The SPDI CGRA executes the loop in the iteration order to keep the dependence among iterations,



(a)



(b)



(c)

Fig. 2. (a) The MII/II generated by the RS compile flow after each sub-phase; (b) The MII/II generated by the REGIMap after each sub-phase; (c) The MII/II generated by the EPIMap after each sub-phase.

which further enables it to handle more than just parallel loops. Along with the compiler developed for the SPDI CGRA architecture, our study gives a quantification of the SPDI CGRA's potential performance relative to the SISP CGRA framework. Besides, we also analyze the impact of hardware design on area overhead and performance.

II. Motivation

The problem of mapping loops onto the SISP CGRA can be partitioned into two sub-phases: scheduling and placing (include routing). Thus, the performance degradation happens in one or both of these phases. We evaluate three SISP CGRA frameworks with different compilers [6]–[8] emerging in recent 5 years by calculating their performance ratio (MII^1/II) generated after each sub-phase (Fig.2). Based on the respective data of the three compilers in Fig.2(a), (b) and (c), the performance of each degrade by 19%, 13% and 20% respectively in the static issue phase and then degrade by 11%, 17% and 13% respectively in the static placement phase. We find that, in addition to the heuristic based placement and routing algorithm, there are 3 main factors making the compilers increase the II due to the SISP CGRA: (1) the inevitable routing node added in the static issue phase, (2) the limitation of using the register, (3) the level constraint in the static issue phase.

A. The Inevitable Routing Node

Fig.3(b) is an data flow graph (DFG) example of a loop. The DFG is a direct graph $D = (V_d, E_d)$ where the nodes

[1]The MII (minimum II), which is the theory peak performance calculated before scheduling and placing, is mainly decided by the resource minimum II (ResMII) and the recurrence minimum II (RecMII).

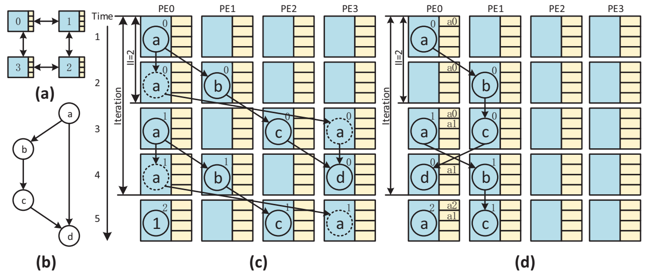


Fig. 3. (a) a 2×2 CGRA with each PE having 4 local registers, (b) an input DFG, (c) a valid mapping routing with PEs, (d) a valid mapping routing with registers.

represent the operations and the edge $(a, b) \in E_d$ iff the output of operation a is an input of operation b. According to the definition in EPIMap [6], the DFG shown in Fig.3(b) is an unbalanced graph because there are two paths from node a to node d and their latency is different. When mapping this DFG to the SISP CGRA, the unbalanced path from node a to d making the result of a computed earlier than c, which means the compiler must safely keep the result of a until c is available.

Most compilers solve this problem by routing with PEs as shown in Fig.3(c). The output register of PE0 in cycle 2 and PE3 in cycle 3 keep the result of operation a, which make the computational resources of PE0 in cycle 2 and PE3 in cycle 3 unable to execute the other operations. Routing with PE leads to the number of the computational resources increase from 4 to 6, which force the compiler to increase the II from 1 to 2 and the performance is reduced 50%. In Fig.2, 6 out of 16 kernels are influenced by this factor.

B. Limitation of using the register

In order to further improve the compiler's mapping ability, REGIMap solve this problem by using registers as shown in Fig.3(d). Instead of keeping the result in the output register, the REGIMap prefers to keep the result in the local register file so that the PE can execute the other operations. Unfortunately, the data stores in the local register file can only be consumed by the PE who owns this register file. This will bring the additional constraint during the mapping phase of the compiler. In Fig.3(d), the compiler has to map node a and node d to the same PE if it chooses to solve the routing problem with registers. The II is also increased from 1 to 2 and the performance is also reduced to 50%. In Fig.2(b) of the REGIMap, 9 out of 16 kernels are influenced by this factor.

C. The level constraint

The level of each node can be defined as:

$$T_{V_d} \ mod \ II \tag{1}$$

The T_{V_d} is the issue time of the node. Under the modulo scheduling executing mode, the level constraint is that the number of nodes at each level must be less than the number of PEs in the CGRA (non-time extended) [6]. If this constraint

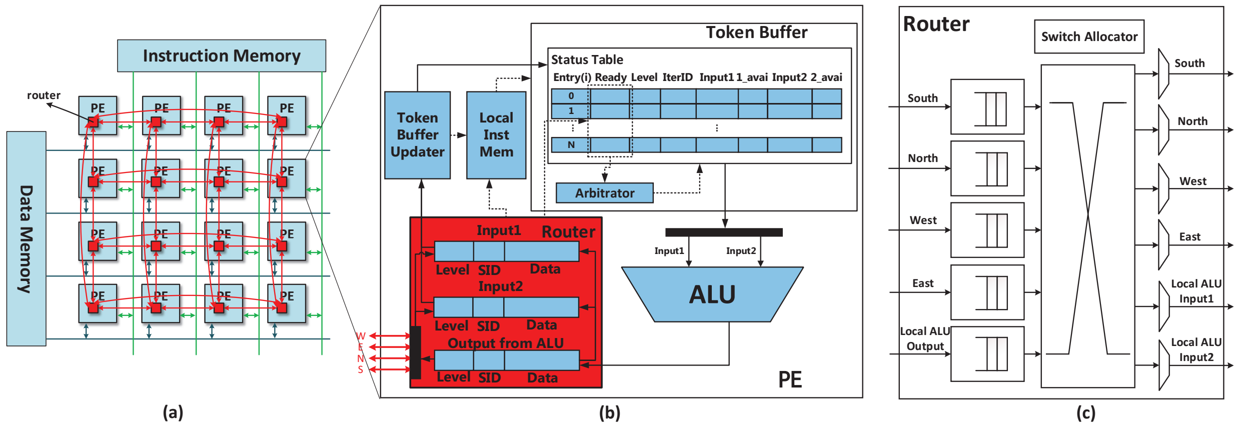


Fig. 4. (a) The SPDI CGRA architecture of 4x4 PE array, the red squares inside each PE represents the routers, they connected through torus topology and form the router network, each of the router binds on a PE; (b) The architecture inside the PE; (c) The high-level diagram of the router.

is not satisfied, the compiler will have to keep adjusting the level by adding the routing node or increasing the II until the level constraint is satisfied. Nearly 50% kernels are influenced by this factor.

These factors push us moving forward to a new CGRA framework that is capable of dynamically schedule the operations. Besides, its PE contains a buffer like module which is more flexible than the register file, thus the above problems that the compiler faces in the SISP CGRA framework can be effectively solved by the hardware in this new framework.

III. Related Work

Our SPDI CGRA framework is designed to tackle the problems of the SISP CGRA in the motivation section. Its execution scheme is inspired by the TRIPS and Wavescalar [9], [10]. They belong to the dynamic dataflow architecture, but they don't use the software pipelining way to execute the program like CGRA does. The TRIPS and Wavescalar's compilers first transform the program IR into the dense block form [11], then map the block onto the core. The goal of the mapping algorithm is to minimize the communication latencies [12], [13] inside the single hyperblock. They are not aimed at accelerating the loops inside the program, but for the powerful general purpose computing to exploit the thread and the instruction level parallelism.

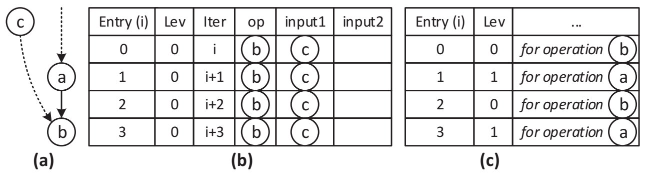
Inside the PE, we use the token buffer to dynamically schedule the operations. Our token buffer based solution is inspired by the SGMF architecture [14], the SGMF architecture is positioned as an energy efficiency design alternative for General-Purpose Graphics Processing Units (GP-GPUs). However, its out-of-order memory access mechanism makes it only capable of handling parallel loops. There are also some relative research works on the token based CGRA and their compilers. For example, the data-flow based reconfigurable architecture for streaming applications and its compiler technique [15], [16], the two-level configuration for FPGA and its mapping algorithm [17], [18]. These works also can't escape from the static issue, static placement execution model. Besides, some of them is designed for the FPGA platform. Paper [19] designs a CGRA with token network, but it is aimed at reducing the control power, which is not focus on optimizing the performance.

IV. The SPDI CGRA

In this section, we present the architecture model of the SPDI CGRA. It consists of a 4x4 PE array, a data memory and an instruction memory (Fig.4(a)). Inside the PE array, PEs are connected through the router network with torus topology and each PE binds a router. Additionally, the PE also contains a token buffer and an ALU. The token buffer can cache the input operands routed from the predecessor PEs, update the attributes of the operation consuming the input operands, and issue the operation to the ALU according to its status. After the token buffer issues the operation to the ALU, the result operand is sent immediately through the router network to its destination PEs with minimum latency. Thus, from the production to consumption of the operand, there are total three steps, and they can work in pipelined manner to improve the throughput of the hardware.

A. The Token Buffer

The architecture of token buffer is presented along with the router and the ALU in Fig.4(b). Inside the token buffer, there is a status table, which is in charge of recording the attributes of the operation. The attributes include: (i) The *Ready* tag recording if the operation is ready to be issued; (ii) The *1_avai* tag and the *2_avai* tag recording if the operation's input operands are available; (iii) The *Level* tag recording the level number of the operation; (iv) The *IterID* tag recording the iteration number that the operation comes from. Besides, the status table also caches the data of the input operands. Every cycle, while a source operation that comes from the new iteration is initiated or new input operands that come from the router's input channel are cached, the status table is updated by the token buffer updater. Additionally, the token buffer also contains an arbitrator, which choose one operation from all the ready operations and sent it to the ALU.



Entry (i)	Lev	Iter	op	input1	input2
0	0	i	b	c	
1	0	i+1	b	c	
2	0	i+2	b	c	
3	0	i+3	b	c	

Entry (i)	Lev	...
0	0	*for operation* b
1	1	*for operation* a
2	0	*for operation* b
3	1	*for operation* a

Fig. 5. (a) A DFG example; (b) A state of the token buffer without classification during the execution; (c) The classification algorithm makes entry 0, 2 allocated for the operation b and entry 1, 3 allocated for the operation a.

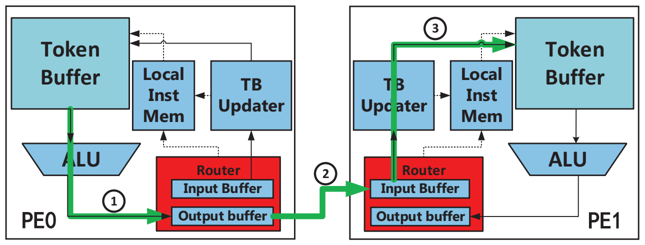


Fig. 6. The datapath of an operand, from it is produced in PE0 to it is consumed by the token buffer in PE1

1) Entry classification: We classify the entries inside the status table so that each entry can only be allocated to the operation with matched level number. The classification aims at avoiding the deadlock. Fig.5(b) presents a deadlock example when the entries are not classified. In the DFG showed in Fig.5(a), we assume the compiler maps two operations, a and b, onto the same PE and the token buffer have 4 entries. The compiler assigns operation a to level 1 and b to level 0. Fig.5(b) shows the deadlock state of this PE's token buffer during the execution. All the entries are allocated for operation b, and they are all waiting for the input operand from operation a. However, there is no entry for operation a, which result in operation a will never be issued and deadlock happens. To tackle this problem, we classify the entries by binding each entry to a matched level number, the entry can only be allocated to the operation under this level number. The matching function is:

$$Lev(i) = i \ mod \ N \tag{2}$$

where $Lev(i)$ is the matched level number. i is the entry number, and N is the number of operations that are mapped to this PE. Fig.5(c) shows the result after the classification, we assume that the level number of the operation a is 1 and b is 0. In this way, the token buffer guarantees the allocation for the operation in any level and avoid the deadlock.

2) The token buffer updater: The token buffer updater is a controller that manipulates the status table. It in charge of allocating the space for the newly arrived input operands according to the matching function and change the state of the status table. Every cycle, whenever there is an input operand reaching the input channel of the router, the updater will allocate an entry for the operation that will consume this input operand. If the operation has been allocated an entry, the updater will find this entry according to the *level* tag and the *IterID* tag, transforms the operand from the router into the status table and set the *1_avai* or the *2_avai* to true.

Additionally, the updater also controls the initiation of the source operation that comes from the new iteration. In the DFG, the nodes without any predecessor are the source operations. The input of these operations are always a constant. Every cycle, the updater will allocate an entry for the source operation as long as there exist a valid entry for it.

3) The arbitrator: The arbitrator is the controller that decides which operation can be issued to the ALU. It first chooses the operations that is qualified to be issued and set their *Ready* bit, then it selects one operation from the ready operations. If an operation wants to be ready, two conditions must be simultaneously satisfied:

- All of its input operands must be available.
- It receives the signal from the router guaranteeing that there are enough valid token buffer spaces (can be allocated for it) in the token buffer of all its destination PEs

This guarantees the cache space for the newly computed data, either in the buffer of the router or in the token buffer of the PE. The arbitrator then chooses the operation with the max level number, if all the operations have the same level number, it will choose the operation with the least iteration number to keep the iteration order of the loop.

B. The router

Fig.4(c) shows the high-level diagram of the router. The router has five inputs and 6 outputs. For the input ports, one for each direction (N, S, E and W), one for the output from its local PE's ALU. All the input ports contains a two entry deep buffer. For the outputs, four ports for each direction and two ports for the input1 and input2 of the token buffer. Inside the router, we use the prioritized switch arbitration, the credit-based flow control and the starvation avoidance technique which are based on the low-cost router design [20]. We choose the low-cost router for several reasons:

- We position the router to accomplish the short distance communication, the longest distance between the producer and the consumer is no more than 2-hops.
- We must consider the trade-off among the performance, the area cost and the power consumption relative to the SISP CGRA framework.
- We expect the data can be effectively sent to the destination with minimum latency.

C. Pipelined datapath

Fig.6 shows the datapath of the operand, from it is produced by PE0 to it is consumed by the PE1. There are total three steps along the datapath, they are pipelined in order to improve the throughput of the hardware. Each step is described as follows:

① In PE0, the operation 0 is issued by the arbitrator and computed by the ALU, the result of the operation (operand 0) is produced, it is cached in the output buffer of PE0's router at the end of this step.

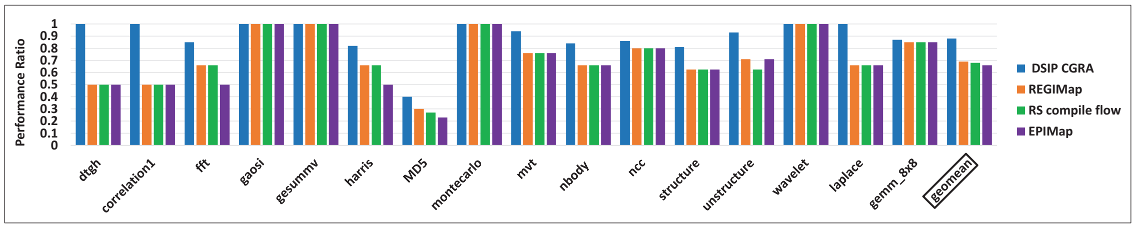


Fig. 7. The performance comparison among the SPDI CGRA framework, the REGIMap, the RS compile flow and the EPIMap under the SISP CGRA framework.

② The operand 0 is sent to PE1 via the routers, data is cached in the input buffer of the PE1's router at the end of this step.

③ The operand 0 is updated by the token buffer updater, it is sent from the router to the token buffer to be the input of its consumer operation.

These three steps can be independent with each other. They can still work in parallel even if there is a stall happens in either of these steps.

V. Evaluation

A. Methodology

We develop both hardware and the software environment to implement the SPDI CGRA framework. For the hardware, we build a C++ runtime system of the SPDI CGRA, which is capable of reflecting the data stream and providing a cycle level performance analysis relative to the SISP CGRA framework. For the software, we design a compiler based on LLVM [21] platform to translate the loop of the C based program to the instruction set of the SPDI CGRA. With the limited space, the detailed mapping approach of the compiler is not included in this paper. We use a heuristic based approach to find the optimized performance under the assumption that every PE can execute its operations without blocking. At last, our runtime system executes the instructions generated by the compiler and obtains the cycle level performance. It is compared with the performance under the SISP CGRA framework.

Based on a fair comparison, we assume both of the SPDI and SISP CGRA have enough memory to hold the instructions as well as variables in the loop and their calculating latency of every operation is within one cycle. Besides, for the SISP CGRA, we use the PE array with PEs connected through torus topology. The size of the register files inside each PE is 16. For the SPDI CGRA, same interconnecting topology is used and each token buffer has 16 entries. We evaluate the performance by calculating the average time interval of the initiation time between two consecutive iterations. It can be computed as:

$$II_{avg} = (T_{finish} - T_{single}) \div (N_{iter} - 1) \qquad (3)$$

where T_{finish} is the time that the SPDI CGRA finish all the iterations of the loop. T_{single} is the latency of a single iteration, it is decided by the latency of the critical path in the DFG of the loop. N_{iter} is the iter number of the loop. In SISP CGRA, the II is the performance metric. Thus we compare the II_{avg} that calculated by our SPDI CGRA with the II generated by the compilers that based on the SISP CGRA. The loops we test are selected from the Mibench [22] and Berkeley 13 [23].

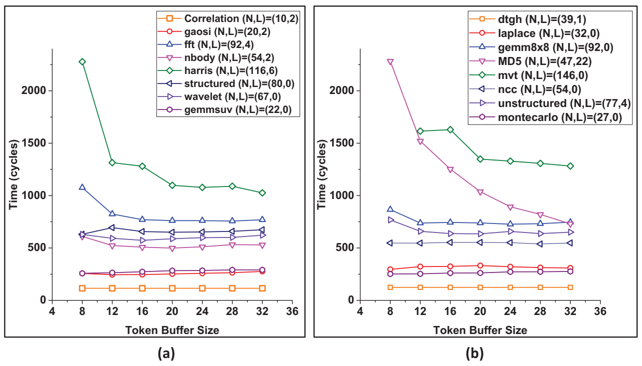


Fig. 8. The performance vs token buffer size in PE from 8 to 32

B. The relative performance

We implement three state-of-the-art compilers based on the SISP CGRA, the REGIMap [7], the EPIMap [6] and the RS compile flow [8]. Fig.7 shows the performance ratio ($MII/II_{avg}, MII/II$) of 16 kernel runs under the SPDI C-GRA framework and the other three SISP CGRA frameworks.

Our evaluation reveals that on average the performance of the SPDI CGRA framework is 1.28, 1.30 and 1.33 higher than SISP CGRA using REGIMap, RS compile flow and EPIMap respectively. According to all the tested kernels, the SPDI CGRA can generate 37.5% optimized performance while the SISP CGRA is 25%. Additionally, compared with the SISP CGRA framework, the SPDI CGRA framework is more advantageous in running all the tested kernels. Furthermore, we believe the performance can be further improved due to the immature technique of our current compiler.

C. Token buffer size influence

To evaluate the effect of the token buffer size on the performance, we run the kernels with token buffer size from 8 to 32. Fig.8(a), (b) separately reveal the running time of 16 kernels, each figure containing 8 kernels. The (N,L) reflects the operation number (N) and the max *latency gap* (L) inside the DFG. The latency gap is the latency difference between the

TABLE I
COMPONENT PARAMETER IN PE OF SPDI AND SISP CGRA

Component Parameter	SPDI CGRA	Component Parameter	SISP CGRA
Token buffer entries (N_m)	8/12/16/20 /24/28/32	register file entries (N_m)	16
bits/entry	82	bits/entry	32
Area/entry(m)	0.005125 (mm^2)	Area/entry(m)	0.002 (mm^2)
Local Inst Mem number of insts (N_i)	64	Local Inst Mem number of insts(N_i)	64
bits/Instruction	128	bits/Instruction	128
Area/Inst (i)	0.008 (mm^2)	Area/Inst (i)	0.008 (mm^2)
Router area (R)	0.016 (mm^2)	Router area (R)	0
Other component (E)	0.05 (mm^2)	Other component (E)	0.05 (mm^2)
PE Area	$N_m \times m + N_i \times i + E + R$		

input operands. Based on the results, there are some factors affecting the performance:

1. For most of the small kernels (N<60), token buffer size has little impact on their performance.

2. Kernels which contain long latency gap (MD5, harris and fft) are easier to be influenced by the token buffer size. Their performances are getting better with the increase of the token buffer size.

3. Large kernels (N>60) may be influenced when the token buffer size is small. However, as the token buffer size increases, their performance will reach a fixed value.

According to this experiment, token buffer size indeed influences the performance of some kernels. However, with the increase of the size, performance will no longer be influenced by the token buffer size. At this time, the factor influencing the performance is the mapping ability of the compiler.

D. Area overhead

We estimate the area size of the PE in SPDI CGRA relative to SISP CGRA. We mainly consider the modules that will consume the area inside the PE such as buffers, FIFOs, entries and registers. The detailed area components we calculate in two architectures are listed in Table I. We think the area of the register and entry are in direct proportion to the number of bits. The area model and the parameter of each component are based on [24]. For the router size calculation, [25] shows that its router size can be optimized to 3% of the TRIPS PE area. The low-cost router [20] further optimizes this size to 2%. Then, we use the area model to calculate the TRIPS PE area in order to get the area size of the router. At last, the area overhead of using different token buffer size is calculated according to the area model. The area overhead is nearly $0.93\% \times N_m - 5.9\%$ for each PE relative to SISP CGRA with 16 local registers, N_m represents the entry size of the token buffer.

VI. CONCLUSION AND FUTURE WORK

This paper proposes the SPDI CGRA model. The token buffer inside the PE and the router network among the PEs array making the SPDI CGRA capable of dynamically schedule the operation and automatically initiate the new iteration. The pipelined stage along the datapath improve the throughput of the hardware. Our evaluation reveals that, with the same token buffer size and the register file size (16) in the PE, our framework can generate on average 1.28, 1.30 and 1.33 higher performance than the REGIMap, RS compile flow and the EPIMap framework with 8.3% area overhead.

REFERENCES

[1] B. Mei, S. Vernalde, D. Verkest, H. De Man *et al.*, “Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling,” in *In Proc. DATE (2003)*, pp. 296–301.
[2] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, “Edge-centric modulo scheduling for coarse-grained reconfigurable architectures,” in *In Proc. PACT (2008)*, 2008, pp. 166–176.
[3] H. Park, K. Fan, M. Kudlur, and S. Mahlke, “Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures,” in *In Porc. CASES (2006)*, pp. 136–146.
[4] L. Chen and T. Mitra, “Graph minor approach for application mapping on cgras,” in *In Proc. FPT (2012)*, pp. 285–292.
[5] A. Fell, Z. E. Rákossy, and A. Chattopadhyay, “Force-directed scheduling for data flow graph mapping on coarse-grained reconfigurable architectures,” in *ReConFig 2014*. IEEE, 2014, pp. 1–8.
[6] M. Hamzeh *et al.*, “Epimap: using epimorphism to map applications on cgras,” in *In Proc. DAC (2012)*, pp. 1280–1287.
[7] M. Hamzeh, A. Shrivastava, and S. Vrudhula, “Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (c-gras),” in *In Proc. DAC (2013)*, pp. 1–10.
[8] Z. Zhao *et al.*, “Resource-saving compile flow for coarse-grained reconfigurable architectures,” in *In Proc. ReConFig (2015)*, pp. 1–8.
[9] K. Sankaralingam *et al.*, “Exploiting ilp, tlp, and dlp with the polymorphous trips architecture,” in *In Proc. ISCA (2003)*, pp. 422–433.
[10] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, “Wavescalar,” in *In Porc. MICRO (2003)*, pp. 291–302.
[11] A. Smith, J. Gibson *et al.*, “Compiling for edge architectures,” in *In Proc. CGO (2006)*, pp. 185–195.
[12] R. Nagarajan, D. Burger, K. S. McKinley, C. Lin, S. Keckler, and S. Kushwaha, “Instruction scheduling for emerging communication-exposed architectures,” in *In Proc. PACT (2004)*, pp. 74–84.
[13] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha, “A spatial path scheduling algorithm for edge architectures,” in *In Proc. ASPLOS (2006)*, pp. 129–140.
[14] D. Voitsechov and Y. Etsion, “Single-graph multiple flows: Energy efficient design alternative for gpgpus,” in *In Proc. ISCA(2014)*, pp. 205–216.
[15] A. Niedermeier, J. Kuper, and G. Smit, “Dataflow-based reconfigurable architecture for streaming applications,” in *In Proc. SoC (2012)*, pp. 1–4.
[16] A. Niedermeier, J. Kuper, and G. J. M. Smit, “A dataflow-inspired cgra for streaming applications,” in *In Porc. FPL (2013)*, pp. 1–2.
[17] M. Allard, P. Grogan, Y. Savaria, and J.-P. David, “Two-level configuration for fpga: A new design methodology based on a computing fabric,” in *In Proc. ISCAS (2012)*, pp. 265–268.
[18] H. Khanzadi, Y. Savaria, and J. P. David, “Mapping applications on two-level configurable hardware,” in *In Proc. Adaptive Hardware and Systems (2015)*, pp. 1–8.
[19] H. Park, Y. Park, and S. Mahlke, “Reducing control power in cgras with token flow,” in *In Proc. Workshop on ODES (2009)*.
[20] J. Kim, “Low-cost router microarchitecture for on-chip networks,” in *In Proc. MICRO (2009)*, pp. 255–266.
[21] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *In Proc. CGO (2004)*, pp. 75–86.
[22] M. R. Guthaus *et al.*, “Mibench: A free, commercially representative embedded benchmark suite,” in *In Proc. WWC (2001)*, pp. 3–14.
[23] K. Asanovic *et al.*, “The landscape of parallel computing research: A view from berkeley,” UCB/EECS-2006-183, Tech. Rep.
[24] S. Swanson, A. Putnam *et al.*, “Area-performance trade-offs in tiled dataflow architectures,” in *In Proc. ISCA (2006)*, pp. 314–326.
[25] P. Gratz, K. Sankaralingam *et al.*, “Implementation and evaluation of a dynamically routed processor operand network,” in *In Proc. NOCS (2007)*, pp. 7–17.