MBDI: A General Paradigm for Mixed Loop Boundary Dynamic issues On CGRAs

***Abstract*—Coarse Grained Reconfigurable Architectures (C-GRAs) have been recognized as desirable platform for repetitive, datapath-oriented and computational intensive applications. One of the primary challenges is to develop compilers that can produce high efficient commands for CGRAs. An important aspect of compiler is the optimization for irregular applications. To this end, this paper makes several contributions: i) Treating loop itself as operations to dynamically control the execution of innermost loop body, which is described as $DBDI$. For the first time in CGRA compilers, we proposed the use of loop operations as a general solution for different kinds of loops. ii) A rule is presented for irregular applications to dispose loop itself and innermost loop body creatively. The arrangement of control flow and placement for loop is amenable. iii) Mixed boundary and dynamic issues(MBDI) method is also introduced to improve the efficiency. MBDI employs the skillful approach that static boundary statically compile,dynamic boundary dynamically execute to reduce both operations and power dramatically.An evaluation is drafted finally under this framework which is accessible to process the data flow with its high quality as well as handcraft. As a result, up to 4x speed up greater than a conventional high-level compile tools. MBDI is capable of achieving the theoretical best performance based on any benchmarks as we practiced. As long as the operation number of loop body is greater than 2, the performance ratio and power ratio remain invincible.**

***Keywords*-Coarse Grained Reconfigurable Architectures; processing element arrays; DBDI; MBDI;**

I. Introduction

To achieve the targets of high power efficiency, most of research on data flow mapping [1], aggressive pipeline [2] and triggered instructions [3] have been exploited in recent years. CGRA occupies the spatial parallelism structure to accelerate classic workload genres. However, it's not adequate to facilitate a few irregular applications with uncertain boundary loop or forcible data/control dependency. Therefore, the performance on speed and energy is limited by compiler, especially for nonuniform programs, even though an efficient hardware is available [4].

During the past decade, much pioneer work has been focused on flattening-based mapping of imperfect loop nests [5], non-kernel execution times [6], graph minor [7], polyhedral model on mapping [8], sub-cycle modulo scheduler [9] and so on. A majority of compilers cater to the process of data flow graph(DFG) mapping and reduction of initiation interval(II) by simplifying programs delicately. Nevertheless, there is still no thorough schemes for all applications. Just as approaches for the optimization of uncertain loop boundary are desired eagerly for a number of applications since it's a margin territory but unavoidable.

In this paper, we take the first step on putting forward dynamic boundary, dynamic issue($DBDI$), an innovative method for data pipelining of loops on processing element(PE) arrays aims at highly simultaneous. Whats more, to lessen operations and economize power, mixed boundary, dynamic issue(MBDI), as another creative technique is feasible to extract more computational throughout. Though it is available to high efficiency and significant performance on speed or power consumption for CGRA, whether can be implemented is critically limited by the compiler technologies. There are three questions must be figured out before an application is allocated to processing element arrays(PEA): i) how to arrange the flow of instructions on loop pipelining to decrease initiation interval(II). ii) where are the operations placed. iii) what is the communication among PEs through router. Actually, these three problems can be summarized to : scheduling, placement and routing. On account of these difficulties, predicted execution [10] has proposed an efficent way for if branch. But this paper presents an opportunity for $loop$.

Owing to the restrictions of previous work based on static loop boundary, much inevitable trouble may be encountered when nonuniform loops with dynamic boundary occurs. Three breakthroughs on CGRA are presented in this paper:
1. Realization of dynamic loop, dynamically execute: The benefits of array topology is to expedite the calculation, particularly the $SIMD$. Whereas, it is a great challenge to accommodate the dynamic boundary loop. In general, pipelining is utilized to parallelize programs, and at a fixed interval, implementing the same operations. To fit the architecture, $DBDI$ takes an important role in scheduling instructions subtly.
2. Establishing a rule for loop mapping: Previously, plenty of CGRA structure have been raised, including HSRA [11], MATRIX [12], RaPiD [13], Tartan [14], MorphoSys [15]. Each of the projects mentioned above requires custom mapping tools that supports a limited subset of structure features [16]. In order to fulfill the gap in previous mapping techniques, we set up a rule to process extra loop instructions. The control flow is also described simultaneously.
3. Extracting an universal principle for mixed loop boundaries: Traditional loops are compiled statically to operate the innermost loop body, extra overhead will be occupied if all loop operations cover the same patterns. It employs much more energy for the static instructions, but only dynamic boundary judgment is demanded. In this case, static compiling the static loop boundary and dynamic com-

mitting the task of dynamic loop orient to any applications moderately. Furthermore, sophisticated mapping of control flow is crucial to the pipelining.

Consequently, we evaluate the $DBDI$ model, and an improved methodology($MBDI$) by simulating several classic kernels as workload. According to 15 benchmarks [17], [18], experiments reveal that the performance on irregular programs is dramatically more efficient than those high-level compile tools as we known currently. Our approach can generate almost 2x average performance ratio compared to traditional solutions of static boundary, static issues($SBSI$).

II. Background and Motivation

PEA is used as the computation vector, composed of 4×4 array(16PEs) and internal connection. Otherwise, a shared memory is provided for data storage. Each PE includes an arithmetic logical unit(ALU), a local register file and a output register, which retains the data for local PE or neighboring PE at next clock. As Fig. 1 depicted, there are two coarse input($IN1$, $IN2$) for computation. It is important to note that input $IN3$ is applied in the control flow, ALU is enable if $IN3 = 0$, otherwise, ALU is idle. Although $IN3$ may be not emerged in most literature, it is necessary to control running in reality. What's more, inter-PEs transfer data through router.

PEA is adopted as coarse grained accelerator, most of work is to study the improvement of flexibility and efficiency. For example, partial predication and full predication [19] or even composite predication [10]. Most of work is focused on $if-else$ structure. But what about loop for, or even $for-if$ or $while-if$?

So far much study has exploited on $if-else$ and innermost loop body [6]. Besides, new parallelism framework is provided for instruction execution and solving the data dependency of innermost loop body by an explicit datapath. However, it's important to note that a headache problem for the acceleration of control flows with loop instruction still remains. It is a new topic for irregular loop applications and worthwhile for us to emerge an efficient approach but not only feasible.

Many irregular control flows for loop in Fig. 2 are displayed. In summary, there are three kinds of irregular loops to be solved: i) It is really difficult for us to arrange the operation of loop mixed because most applications are only focused on regular statements. Such as $for-if$ mixed in bucket sort, $while-if$ mixed in Fibonacci Search, execution operations mixed in autocr. ii) Loop itself is treated as operations. Such an instruction is often adopted to the kind of sort algorithm, for example in Binary Search. iii) Loop is not certain ahead of run-time. As we listed, the initialization value of loop is uncertain in SPMV and LU, loop boundary is mutable in DFS, loop step is float in Knapsack Problem. Since only static loop operations are considered for most high-level synthesis tools. In other words, if there is a blurry

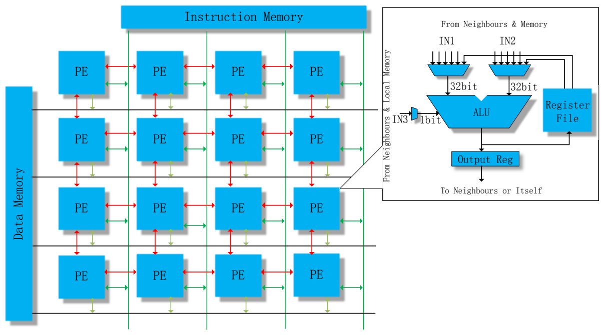


Figure 1: The structure of 4x4 PEs inside CGRA computing model, composed of ALU, output register, local register file.

instruction block, this application may be not appropriate for CGRA, application scenarios to a great extent is limited. To unite statement, it will be named as dynamic loop in this paper if the loop boundary is uncertain. Otherwise, static loop is accessible for fixed boundary.

All in all, so many irregular applications are hard to manage, but they are general in classic algorithms. Hence a novel scheme is valuable to address the full opportunities. Although we prefer to present the problem iii) in the paper, other situations are amenable to be processed in the same methodology. Details on the previous methods are discussed in next section, then a high efficient one is carried out later.

III. Related Work

As we mentioned earlier in section 1, three prominent questions must be overcomed in compiler. Furthermore, it is related to map, scheduling and routing in loop data flow graph (DFG). It is recognized that the synthesis tools have significant influence on the performance for general applications. Most research has focused on how to mapping and schedule for innermost loop body, including branch instructions [20]–[25]. Whereas, the irregular applications are not rare for general kernel, especially in search and sort domains. This paper takes the first step to discuss these questions, a traditional approach we named as static boundary, static issue(SBSI), which is used to make a contrast. In terms of simple dynamic boundary in a loop, the maximum loop times is adopted to the whole loop boundary. It is helpful for us to find the best schemes currently even though it is not familiar for us to study the best program in this new area so far. Perhaps based on one of the feasible methods, we will present a novel method closed to ideal situation by handcraft.

As far as we concerned, SBSI is an optional approach to dispose uncertain boundary loop. However, several drawbacks are clear for SBSI. It is feasible that only the kind of application whose boundary are available for us to obtain in compile time not in run-time. At the same time, many invalid iteration are also executed with branch instructions

Bucket Sort	DFS	Binary Search	Autocr
1: for(lv1 = r1; lv1< c1;lv1+= s1) 2: if(arr1[lv1]!=0) 3: for(lv2 = r2; lv2< arr2[lv1];lv2+= s2) // loop body	1: for(lv1 = r1; lv1< c1;lv1+= s1) 2: for(lv2 = r2; lv2< c2;lv2+= s2) 3: for(lv3 = arr1[lv2]; lv3< arr1[lv2+1];lv3+= s3) // loop body	1: initialize lvi,lvj 2: while(1) 3: while(arr[++lvi] < c) ; 4: while(arr[--lvj] > c) ; // loop body	1: for(lv1 = r1; lv1< c1;lv1+= s1) 2: v1 = 0; v2 = c1 – lv1; 3: for(lv2 = 0; lv2< v2;lv2++) // loop body
Fibonacci Search	SPMV	Knapsack Problem	LU

1: // Processing variable v1,v2,v3, array arr1,arr2
2: while(arr1[v1]>0)
3: if(arr2[v2] < v3)
4: v2 += arr1[--v1];
5: // Other statements
	1: for(lv1 = r1; lv1< c1;lv1+= s1)
	2: for(lv2 = arr[lv1]; lv2< c2;lv2+= s2)
	// loop body
		1: for(lv1 = r1; lv1< c1;lv1+= s1)
		2: for(lv2 = c2; lv2 >= C3;lv2-= arr1[arr2[lv1]])
		// loop body
			1: initialize lvi
			2: while(1)
			3: if(lvi > v) break;
			4: lvi ++; // loop body 1
			5: for(lv2 = lvi; lv2 < c; lv2++)
			// loop body 2

lv: the loop variable   r: initialization reset value i   *c*: the constant value   *s*: the step value for loop   *v*: variance   *arr[i]: the ith v*alue in array

Figure 2: Kinds of loop function in benchmarks.

to guarantee its accuracy, which are not efficient because much more run time and power are required. As a result, irregular loop applications contradict with the demand of high efficiency and low power in CGRA.

Proper hardware and software combination allow us to systematically solve the irregular loop applications. Then, the concept of $DBDI$ are explained firstly, and the accuracy are important to be verified too. Whats more, more valid strategies are presented to promote II and eliminate operations as possible as we can. Finally, an unified algorithm is achieved naturally before we show the experimental results.

IV. DBDI

In this section, the general uncertain loop structure is simplified in terms of DFG to decrease the II. In addition, details on the aspect of $DBDI$ is represented exactly step by step. Furthermore, the bottleneck of $DBDI$ on the II is reduced to 2, or even 1.

A. Overview

DFS kernel is chosen to describe the problem due to its classic and characteristic with three-level loop. As shown in Fig. 3(a), indentation indicates a program block and each operation is tagged from $a \sim k$. we simplify inner loop body as only one line issue g to focus insight on the feasibility and accuracy of $DBDI$ owing to the mature technique of mapping, scheduling, and placing for innermost loop body already [4], [5], [7], [8], [16], [20], [22]–[24], [26], [27]. The loop boundary of the two outermost levels is static, then compiler runs a such application with fixed step, times and reset value. Above all, it is the characteristic of SBSI.

In the following sections, data flow graph(DFG) and an improved-DFG are introduced. We take the first step of the mapping and scheduling for the applications with dynamic loop boundary.

B. Improved-DFG

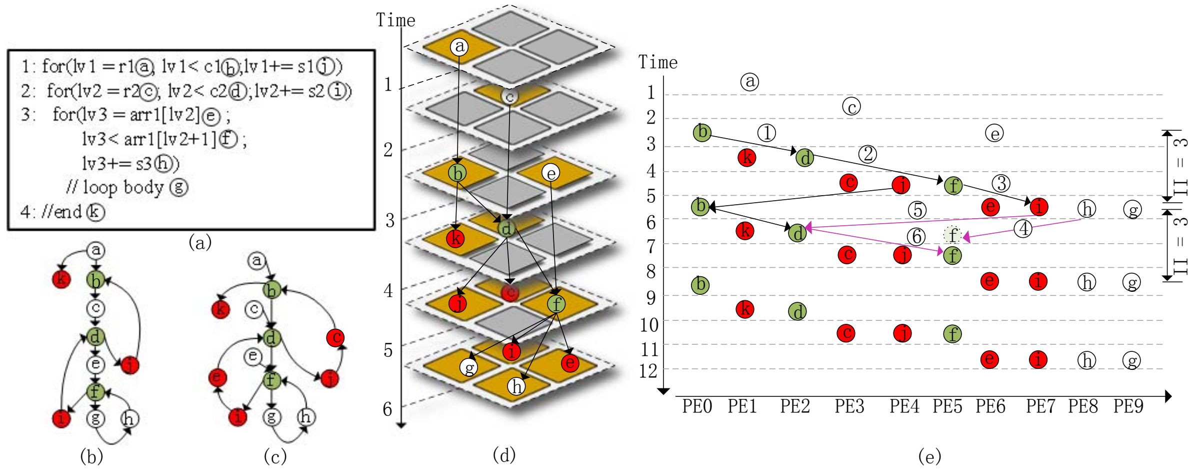
Fig. 3(b) drafts the DFG, which is a directed graph $D = (V_d, E_d)$ where nodes represent for issue and Ed stands for the link between nodes, and arc $(a, b) \in E_d$ iff the output of operation a is an input of operation b [1]. The directed graph expresses that b is executed after a completed, which indicates the time scale of nodes. According to the order of execution in Fig. 3(a), program begins at a, then judges at b, line 2 is accessible if the judgment is true, or line 4 is on the line. Lastly, the loop variable j is updated.

Improved-DFG is displayed ahead in Fig. 3(c), given graph $D = (V_d, E_d)$, let n be the number of nodes in Vd, that is, $n = |V_d|$. A node is described by $n_i = (v_i, e_i) \in D, e_i = (i, r), 1 \leq \forall i \leq n$, let $C = (V_c, E_c)$ be the time extended CGRA(TEC) [1], let C_i be a subset of C, ie, $C_i \subseteq C$. Define $S = \{S_1, S_2, \cdots, S_n\}$ is the set of sub graph, S_i is consist of adjacent nodes. Then, we give the definition of **critical path**,

Definition 1. *Assuming that there is a node $n_z = (v_z, e_z) \in D, e_z = (z, r)$, $1 \leq \forall z, \forall r \leq n$ at least has two pre nodes that satisfy $\exists n_p = (v_p, e_p) \in D, e_p = (p, z), 1 \leq \forall p \leq n$, and $\exists n_q = (v_q, e_q) \in D, e_q = (q, z), 1 \leq \forall q \leq n$. Then $\exists S_{pz}, \{n_p, n_z\} \subseteq S_{pz}, \exists C_{pz}, C_{pz} \subseteq C, 1 \leq \forall pz \leq n$, at the same time, $\exists S_{qz}, \{n_q, n_z\} \subseteq S_{qz}, \exists C_{qz}, C_{qz} \subseteq C, 1 \leq \forall qz \leq n$, the longest path $max\{C_{pz}, C_{qz}\}$ is the **critical path**.*

In Fig. 3(b), the red circle is selected if the judgment(green circle) is false. From *Definition 1*, the green circle is the node(n_z) that we look for. There are at least two nodes referred to it, thus two datapaths are achievable. As for PEA, it operates in the form of pipelining, the interval of the next loop is noted as II generally. And designers always try every effort to cut down it. With regards to node n_z, we choose the critical path to decide II. Accordingly, a C is available to TEC.

As above analyzed, we long to reduce the length of critical path to reduce II. Because no other operations except $for/while$ between line 1 and line 3 in Fig. 3(a). So the reset nodes(a,c,e) can move to another path in Fig. 3(c). It is beneficial to compress II, thanks to tuned nodes(a,c,e) owning concurrency after the startup of loop. It is clearly shown in Fig. 3(d), c is excuted with f at time 5, and e is available with g, h. It is easy to verify that the functionality remains when DFG is converted into Fig. 3(c).



```
1: for(lv1 = r1ⓐ, lv1< c1ⓑ;lv1+= s1ⓙ)
2:  for(lv2 = r2ⓒ; lv2< c2ⓓ;lv2+= s2ⓘ)
3:   for(lv3 = arr1[lv2]ⓔ;
       lv3< arr1[lv2+1]ⓕ;
       lv3+= s3ⓗ)
      // loop body ⓖ
4: //end ⓚ
```

Figure 3: (a) Loop program flow of DFS, (b) DFG of loop, (c) simplified DFG, (d) scheduling of the instructions on the 2×2 PEs, which is time extended, (e) mapping of three-layer loops, the minimum II is 3.

C. Dynamic Boundary Dynamic Issue

As shown explicitly in Fig. 3(d), because the space is limited, we employ 2×2 PEA to present TEC, which displays the scheduling of DFG. And the grey blocks may be utilized for another issues in a new iteration. In Fig. 3(d), node d is executed at $t = 4$. Two paths are accessible($a-b-d$ and $c-d$), it is clear that $a-b-d$ is the critical path. In Fig. 3(e), operations are unfolded explicitly.

To illustrate the time scale and mapping of DFG elaborately, we allocate each issue in a single PE as disposed in Fig. 3(e). The placement of nodes is determined by its dependency. Arrows in figure represents different paths. For example, two purple arrow lines reveal the feasible execution paths of node f. The earliest time of operation for path 4 is $t = 7$, but path 6 is $t = 8$. Therefore, path 6 represents for the critical path.

Above all, the minimal II is 3 through improved-DFG. However, it is not the best one if the number of operations in innermost loop body is smaller than 3. Next section will address this problem directly.

D. Improved-DBDI

Apart from scheduling, placing and routing of data transfer on PE, the controlling is also on the agenda. Each PE owns a control fine port($IN3$) which is responsible for the activation of ALU. Thus, not only routing for data transaction is in need, but also routing for control is significant. We consider a **rule** for the arrangement of control flow and data dependency in the loops:

Definition 2. *A **rule** is a promise to control the execution of issues from outside to inside, but II of TEC is transferred from inside to outside.*

Considering the judgment of outer loop node $n_{z\_o} = (v_{z\_o}, e_{z\_o}) \in D$, $e_{z\_o} = (z\_o, z\_ro)$, $1 \leq \forall z\_o, \forall z\_ro \leq n$ and the judgment of inner loop node $n_{z\_i} = (v_{z\_i}, e_{z\_i}) \in D, e_{z\_i} = (z\_i, z\_ri), 1 \leq \forall z\_i, \forall z\_ri \leq n$, there is an adjacent link between n_{z_o} and n_{z_i}, where an arc $e_{z\_o} = (z\_o, z\_ro)$ is the bridge of n_{z_o} and n_{z_i}. As we expected, loop is initialize form outside to inside, direction of the arc is pointed to inner loop nodes. Therefore, the control flow starts at outer loop, which caters to the operation pattern of pipelining. Communication for control is associated with Fig. 4(a).

In fact the dependency among loop variables, such as $lv1$,$lv2$,$lv3$ in Fig. 3(a), is likely to be deemed as stack. The stack is established $S_t = \{S_{lv1}, S_{lv2}, S_{lv3}\}$, while judgment nodes $n_{z\_lv1}, n_{z\_lv2}, n_{z\_lv3} \in S_{lv1}$; $n_{z\_lv2}, n_{z\_lv3} \in S_{lv2}$; $n_{z\_lv3} \in S_{lv3}$, besides $S_{lv1}, S_{lv2}, S_{lv3} \subseteq S$. The order of running is elucidated as follows. Firstly, sub graph S_{lv3} is updated. If program skips out of the inner block, then sub graph S_{lv2} is updated. Finally, if n_{z_lv2} output false, sub graph S_{lv3} is involved. To sum up, when the program launch, push order of the stack is $S_{lv1} - S_{lv2} - S_{lv3}$. At running time, the update order is $S_{lv3} - S_{lv2} - S_{lv1}$. The last in first out(LIFO) stack S_t contributes to the execution of loop, which is in accord with the data dependency. It is closely involved in purple line in Fig. 3(e).

The overwhelming majority of performance preserves on II [1]. It is clarified in Fig. 3(e), whose $II = 3$. There is a bottleneck on the inner loop body of which $II < 3$ owing to the launch gap increase. Indeed, the key of reducing II is to allow next iteration launch earlier. Accordingly, one way is to start next iteration at $t = 7$ even though there are two paths(4 and 6) for next node f. We will double judge for operations e to decrease the II. Another way is to start a new iteration from inner to outer at time 7. It is prominent to state that this approach is launched from inner to outer, but the former only judge twice for innermost for. Thanks to the former may catch index exception out of array if path 6

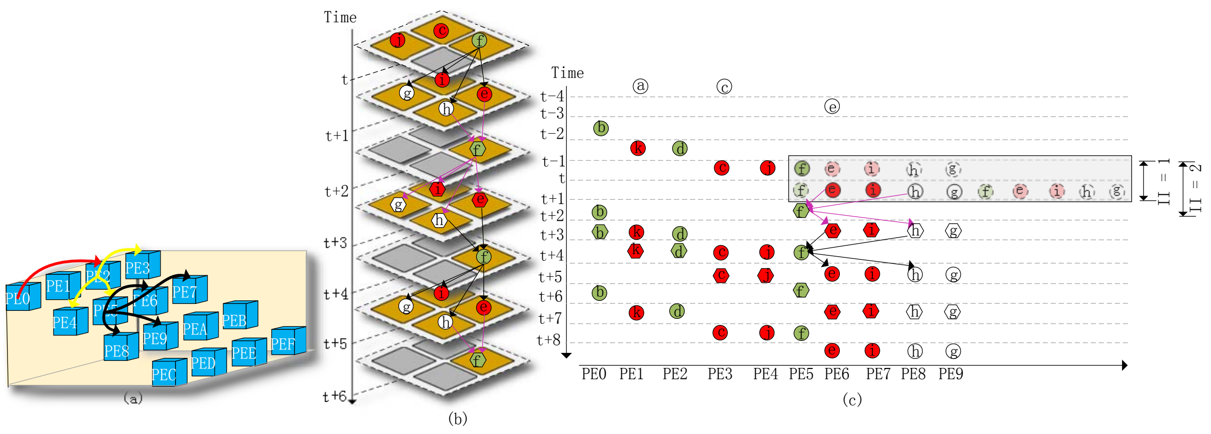


Figure 4: (a) The dependency of control flow among PEs, thus input $IN3$ of PE2 is from PE0. In the same way, $IN3$ of PE3/4/5 is from PE2, (b) on 2×2 PEA, cross execution of inner loop, which is tiled on TEC, (c) mapping and time scale of three-layer loops, the II of cross execution is 2, the minimal value is 1 extremely.

is access. Then only the later way is available. To conclude, alternate crossover execution is dedicated with circle and hexagon in Fig. 4(c). And the minimum II is 2.

To analyze time scale, assume that there are i, j, k layers for three nested loops separately. In addition, dynamic boundary of k variance is a sequence array K, inner loop body is m, then the number of operands(O) for SBSI is:

$$O_{SBSI} = \sum_i \sum_j \max(K) \cdot m \tag{1}$$

The reduced loops(L) of $DBDI$ compared to SBSI is profiled,

$$\Delta L = \sum_i \sum_j (\max(K) - k) \cdot m, k \in K \tag{2}$$

On the one hand, $DBDI$ takes less loops. On the other hand, it adds extra operands(PE0∼PE8) of loop in Fig. 3(e). The productivity of total operands may not cut down because of the trade of less loop and more issues. It brings in additional 9 issues (3 issues each loop) for 3 layers loop generally. So we must consider this function if we want to attain the better efficiency all the time.

$$\Delta O = \sum_i \sum_j k \cdot (m+9) - \sum_i \sum_j \max(K) \cdot m, k \in K \tag{3}$$

If K is a continuous sequence $[1, 2, \cdots, N]$, and index j is from 0 to $N-1$, then formula (3) can be simplified as,

$$\Delta O = \sum_i [(m+9) \cdot \frac{N(N+1)}{2} - m \cdot N^2] \tag{4}$$

It is accessible to be verified that only $m > 9$, $DBDI$ is meaningful. Or most operations are focused on the appendix. It is reflected on $autocr$ kernel in Fig. 5(b).

In general, the number of instructions in most inner loop body is larger than 2, so the II lies on explicit application. An optional approach is still assisted in when there is only one issue in the loop body. Branch prediction [10], [28] is used to execute issue ahead of condition, which is tagged as dashed cycles by grey rectangle in Fig. 4(c). On the whole, II is decreased to 1 with 2x hardware.

E. Mixed Boundary Dynamic Issue

Generally, the interpretation of dynamic boundary dynamic executing the loop function is exploiting the control flow to lesson data flow. The method is efficient but not rare for us in $Turing\ Machine$. $DBDI$ has a sensual message to reduce the execution time by lessening times of iterations at the same time. If the boundaries of all loops are uncertain, the extra issues for loop itself is necessary. If m describes the number of commands for the innermost loop body, dm counts the dynamic loop and sm illustrates the amount of static loops. As we know loop body m and dynamic loops dm must be taken in run-time, but the static loops can be processed in compile time. If static loop is also taken into consideration in run-time, PE utilization factor is exhibited by the ratio $\frac{m+dm}{m+dm+sm}$. It is clear that the ratio is 100% if all loops are dynamic.

However, most of applications owns mixed loop boundary according to Fig. 2. Then the utilization factor is lower than 100%, since static loops sm exists in the denominator. According to the analysis above, it is easy to figure out that $DBDI$ always accelerate programs of which the for judgment can not determine at the compile time.

To solve the mixed boundary of static and dynamic, mixed boundary dynamic issues($MBDI$) approach is proposed. It is beneficial for us to compile the static loop statically and execute the dynamic loop in real time. So if outer two levels are compiled statically, only extra 3 operations are demanded. As the experiments presented in section 5, the performance ratio increases as the utilization factor grows.

As algorithm 1 revealed that the static optimizations is optional for dynamic loops. Since $DBDI$ will always accelerate dynamic loops, time priority(TP) is flexible to accelerate programs when we are not concerned about the resource or power efficiency sometimes. If all programs are consist of static loops, line 3 will process it in a conventional way. Otherwise, we will make use of $MBDI$ certainly. Generally, the structure is $static\ loop - static\ loop - dynamic\ loop$ ad DFS, dynamic loop is executed directly by MBDI in line 18. In order to improve the robustness, $static\ loop - dynamic\ loop - static\ loop$ samdwich structure is taken into consideration too. For high efficiency, the inner $static\ loop$ is tiled generally, then the loop layer is decreased to 2. However, if the innermost loop body m is too long, it will put enormous stress on compiler. Thus unrolling is useful in line 14, or we will treat the inner $static\ loop$ as dynamic loop in line 7∼11, which will occupy more PEs. Then innermost loop body is committed in line 20.

For $DBDI$ always achieves the best performance at time scale according to equation (2) related to $SBSI$. If we are uncertain when to compile applications in $SBSI$ and when to compiled it in $MBDI$, we will trade off to make it more intelligent. As we analyzed above, if loop body m is 1, II of predication scheduling only takes 1 to avoid worsen programs in line 5, which is shown in algorithm 2. Whereas, if the number of iterations are too large after evaluating in line 2, $SBSI$ may be more simple due to resource limited. In this part, we illustrate that $MBDI$ is always beneficial if $m > 3$, so line 16 is accessible. Otherwise, a selection must be done from $DBDI$ and $SBSI$, just as saving time or saving power of which we prefer. We evaluate it by the computations of lessened operations and lessened iterations in line 8∼9. What's more, the degree of preference is adjusted with coefficient α in line 10 .

To lesson the operands as possible as we can, MBDI is devoted to compile mixed boundary with static and dynamic. As we concluded above, loop itself is also considered as operations in PEA, which makes it amenable to take control dependency with loop body. As well as providing the paradigm of scheduling, loop tiling skill is suitable for some irregular applications.

F. Space efficiency of placement

It is clear that each type of operations is allocated to a single PE. By this way, the analysis of TEC, scheduling and routing is unambiguous and efficient. Nevertheless, the space efficiency of placement is really not lofty. In order to describe this problem appropriately, the space utilization ratio (SUR) is illustrated,

Definition 3. *Let T be the period where performs the same operations on PE. And activated time is t, $\forall t \leq T$. Then, $SUR = \frac{t}{T} \times 100\%$.*

Algorithm 1 MBDI Static Optimization Algorithm

```
Require: Time priority $\mathcal{TP}$
Input: Total loop info $TL$, $Mixed$ ,$TDepth$,loop itself info
    $Li$, loop body info $Bo$, syntax block $B$
Output: transfer static to dynamic $S2D$, loop tiling $LT$,
    static/dynamic/predict scheduling are $SS/DS/PS$
 1: while $B$ in Queen do
 2:   if !$Mixed$ then
 3:     output $SS$
 4:   else
 5:     if $\mathcal{TP}$ then
 6:       if $B$ is dynamic loop and $B.depth < TDepth$
  then
 7:         if $Bo.nu > N1$ then
 8:           $dy\_nu+ = TDepth - B.depth$;
 9:           while inner loop $B_i$ in Queen do
10:             output $S2D$
11:           end while
12:         else
13:           while inner loop $B_i$ in Queen do
14:             output $LT$
15:           end while
16:         end if
17:       else if $B$ is loop then
18:         output $DS$
19:       else
20:         output $SS$
21:       end if
22:     end if
23:   end if
24: end while
```

Algorithm 2 MBDI Dynamic Optimization Algorithm

```
 1: while $B$ in Queen do
 2:   if $Bo.nu < 2$ and $TL.maxLoop > N2$ then
 3:     output $SS$
 4:   else if $Bo.nu < 2$ then
 5:     output $PS$
 6:   else
 7:     if $Bo.nu \leq 3 \times TL.loop$ then
 8:       $\Delta O = lessonO()$;
 9:       $\Delta L = lessonL()$;
10:       if $\alpha \times \Delta O > \Delta L$ then
11:         output $DS$
12:       else
13:         output $SS$
14:       end if
15:     else
16:       output $DS$
17:     end if
18:   end if
19: end while
```

Therefore, the average of SUR(ASUR)can be summed up as follows:

$$ASUR = \frac{\sum_{i=0}^{n} SUR_i}{n} \tag{5}$$

Where n is the number of PE used.

Consequently, formula (5) is used to indicate the space efficiency of placement. Pursuing investigation into ASUR is 33.3% for Fig. 3(e) because each SUR of PE is the same. In addition, we provides two feasible methodology to promote ASUR.

- It is possible to place PEs together, which is related to the same for or $while$ sentence. Such as PE0, PE1 and PE4 can be placed together in Fig. 4(c). While hexagon operations are placed in a new array of PE, for example PE10∼PE20 if PEs are enough. Hence, it can be verified that operation a,b,j in Fig. 3(a) will never occupy PE in the same time. In this way, $ASUR = 50\%$.
- Another way is just seem to Fig. 4(c). And PE6/7/8/9 is capable of emerging into PE5, ASUR is 56%. It is simple to prove that ASUR is 50% for each PE in Fig. 4(c), but there is a combination in PE5, improving ASUR.

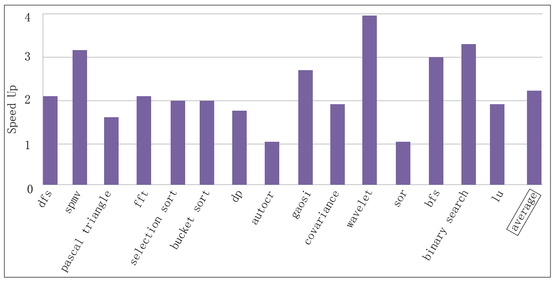
V. Evaluation

Both hardware and software platform are developed to evaluate the performance of dynamic loop. In hardware, 4×4 homogeneous PEA is adopted in our CGRAs, two local registers are provided, which is reachable to itself or neighbouring PEs. We assume that memory is enough and will take 1 instruction for store or load. In software, we developed a new language GRC based on C to specify irregular loops in source. The irregular applications is selected from SPEC2006 benchmarks [18], Berkeley 13 [17] and classic sort and search algorithms. $SBSI$ is employed for reference, we admit that it is not an elegant approach, but it don't prevent us from performing the best methodology.

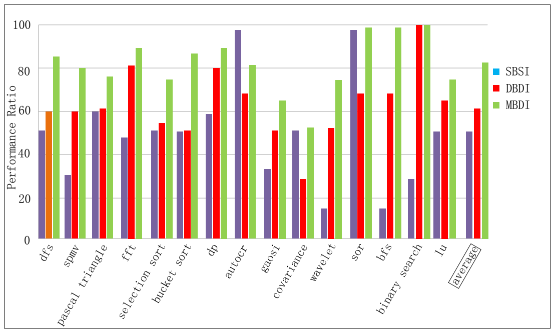
In conclusion, if the number of operations in loop body is greater than 3, may be 2 compared to SBSI because it also takes 1 operation to judge, MBDI will always leads to the optimal property. It's scalable and robust to other platform owing to flexible algorithm.

A. Acceleration of DBDI and MBDI

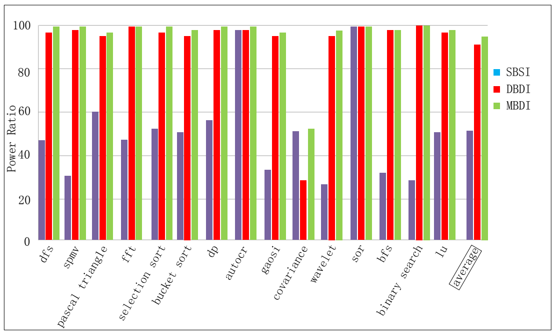
Alternatively, both $DBDI$ and $MBDI$ are feasible to speed up the program. In Fig. 5(a), the speed up of 15 applications is larger than 1 as we expected. Specification of speed up is calculated from the time division of $DBDI$ and SBSI. And MBDI has the same effects as $DBDI$. The performance differs greatly in specific application domain. For $wavelet$ and $binarysearch$ algorithm, the number of dynamic loop layers is larger than others, then $DBDI$ plays an more important role in programs. But the reason why $autocr$ and sor kernel don't achieve commendable performance is that the loop body m is too small. So it is not obvious for the improvement of total performance.



(a) The speed up of DBDI is up to 4x than SBSI based on 15 benchmarks.



(b) The comparison of performance ratio among SBSI,DBDI and MBDI.



(c) Power efficient is very high for DBDI and MBDI.

Figure 5: Evaluation for speed, performance and power.

B. MBDI Improves Efficiency from DBDI

It is nature that $DBDI$ always achieves a better time performance than SBSI according to Fig. 5(a) since only a few loops are valid in fact. What's more, the relative performance (in percentage) is described by the division of Minimum Operands and Operands, i.e. $\frac{MO}{O}$. In the same way, specification of relative power is noted by $\frac{MP}{P}$. Therefore, the higher value is much closer to the ideal one.

As depicted in Fig. 5(b), MBDI almost stands for the best situation, only necessary operations are carried out. But $DBDI$ don't perform well in $autocr$,$covariance$ and sor, as a matter of fact the loop body m is too small. Thus most of resource are cost in PE0∼PE8 in Fig. 3(e). At the same time, $autocr$,$covariance$ and sor manifest not very well in power as plotted in Fig. 5(c). The reason why MBDI are always closed to the ideal performance is that it takes only necessary iterations and execute dynamically at crucial time. Sometimes MBDI's performance don't over 90% in Fig. 5(b), because there are many dynamic loop levels but small innermost loop body. Therefore, the dynamic loop operations are essential and the minimum operands are overidealizing.

It is observed obviously that $DBDI$ and MBDI are always accelerate parallelism program. A principle is also concluded that if the number of inner body is lager than $2(m > 2)$ for single dynamic loop layer, MBDI are always get the best performance and low power as displayed.

VI. Conclusion

As compiler technologies and advanced mappings skills take important role in CGRA, extracting sophisticated schedule and elegant routing for data communication is feasible. To fill the gap for irregular loop applications, this paper takes loop itself as operations creatively. What's more, a scalability strategy is accommodated to address this challenge. It is verified that the II is only 2 generally but 2x speed up in average. From the experiments, our scheme leads to significant performance improvement and is closed to the desirable one.

References

[1] M. Hamzeh *et al.*, "Epimap: Using epimorphism to map applications on cgras," in *DAC*. IEEE, 2012, pp. 1280–1287.

[2] Z. Li, L. Liu *et al.*, "Aggressive pipelining of irregular applications on reconfigurable hardware." ACM, 2017, pp. 575–586.

[3] P. others, "Triggered instructions: A control paradigm for spatially-programmed architectures," vol. 41, no. 3. ACM, 2013, pp. 142–153.

[4] Y. Kim *et al.*, "Operation and data mapping for cgras with multi-bank memory," vol. 45, no. 4. ACM, 2010, pp. 17–26.

[5] J. Lee *et al.*, "Flattening-based mapping of imperfect loop nests for cgras?" in *CODES+ ISSS, 2014 International Conference on*. IEEE, 2014, pp. 1–10.

[6] Y. Kim *et al.*, "Improving performance of nested loops on reconfigurable array processors," *ACM TACO*, vol. 8, no. 4, p. 32, 2012.

[7] L. Chen *et al.*, "Graph minor approach for application mapping on cgras," *ACM TRETS*, vol. 7, no. 3, p. 21, 2014.

[8] D. Liu, S. Yin, L. Liu, and S. Wei, "Polyhedral model based mapping optimization of loop nests for cgras," in *DAC, 2013 50th ACM/EDAC/IEEE*. IEEE, 2013, pp. 1–8.

[9] Y. Park *et al.*, "Cgra express: accelerating execution using dynamic operation fusion." ACM, 2009, pp. 271–280.

[10] J. Wang *et al.*, "Acceleration of control flows on reconfigurable architecture with a composite method." ACM, 2015, p. 45.

[11] W. Tsu *et al.*, "Hsra: high-speed, hierarchical synchronous reconfigurable array," in *1999 ACM/SIGDA*. ACM, 1999, pp. 125–134.

[12] E. Mirsky *et al.*, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources." in *FCCM*, vol. 96, 1996, pp. 17–19.

[13] Hartenstein *et al.*, "Field-programmable logic smart applications, new paradigms, and compilers," 1996.

[14] M. a. Mishra, "Virtualization on the tartan reconfigurable architecture." IEEE, 2007, pp. 323–330.

[15] H. Singh *et al.*, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," vol. 49, no. 5, pp. 465–481, 2000.

[16] S. Friedman *et al.*, "Spr: an architecture-adaptive cgra mapping tool," in *ACM/SIGDA*. ACM, 2009, pp. 191–200.

[17] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from berkeley," UCB/EECS-2006-183, Berkeley, Tech. Rep., 2006.

[18] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH*, vol. 34, no. 4, pp. 1–17, 2006.

[19] A. Parashar *et al.*, "Efficient spatial processing element control via triggered instructions," *IEEE Micro*, vol. 34, no. 3, pp. 120–137, 2014.

[20] M. Ahn *et al.*, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," vol. 1. IEEE, 2006, pp. 6–pp.

[21] G. Dimitroulakos *et al.*, "Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures." IEEE, 2006, pp. 10–pp.

[22] S. Friedman *et al.*, "Spr: an architecture-adaptive cgra mapping tool," in *ACM/SIGDA*. ACM, 2009, pp. 191–200.

[23] Y. Guo *et al.*, "A pattern selection algorithm for multi-pattern scheduling." IEEE, 2006, pp. 8–pp.

[24] A. Hatanaka *et al.*, "A modulo scheduling algorithm for a coarse-grain reconfigurable array template." IEEE, 2007, pp. 1–8.

[25] J.-e. Lee *et al.*, "Compilation approach for coarse-grained reconfigurable architectures," vol. 20, no. 1, pp. 26–33, 2003.

[26] H. Park *et al.*, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures." ACM, 2008, pp. 166–176.

[27] J. W. Yoon *et al.*, "A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures," *IEEE Transactions*, vol. 17, no. 11, pp. 1565–1578, 2009.

[28] S. Mahlke *et al.*, "Compiler synthesized dynamic branch prediction." IEEE Computer Society, 1996, pp. 153–164.