Register File Organization for Coarse-Grained Reconfigurable Architectures:

Compiler-Microarchitecture Perspective

by

Dipal Saluja

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2014 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair

Yann-Hang Lee

Carole-Jean Wu

ARIZONA STATE UNIVERSITY

August 2014

ABSTRACT

Coarse-Grained Reconfigurable Architectures (CGRA) are a promising fabric for improving the performance and power-efficiency of computing devices. CGRAs are composed of components that are well-optimized to execute loops and rotating register file is an example of such a component present in CGRAs. Due to the rotating nature of register indexes in rotating register file, it is very challenging, if at all possible, to hold and properly index memory addresses (pointers) and static values. In this Thesis, different structures for CGRA register files are investigated. Those structures are experimentally compared in terms of performance of mapped applications, design frequency, and area. It is shown that a register file that can logically be partitioned into rotating and non-rotating regions is an excellent choice because it imposes the minimum restriction on underlying CGRA mapping algorithm while resulting in efficient resource utilization.

DEDICATION

To my parents and my sister who have always encouraged me to pursue my dreams and work towards the betterment of mankind.

ACKNOWLEDGMENTS

I am thankful to Prof. Aviral Shrivastava for giving me an opportunity to work with him on this project. He has always been there as a guide and a source of immense inspiration. He helped me explore my true potential.

I am thankful to Prof. Yann-Hang Lee for being a wonderful teacher. His methodological and student friendly approach to teaching not only motivated me to gain technical skills but also helped me develop a detail oriented mindset and life skills.

I am thankful to my friend and colleague Mahdi Hamzeh for always being there to help me look at the big picture and help me with the project.

I am thankful to my dear friends Hitesh Khunti, Vinayak Kumar, Mohit Shah and Saurabh Jaluka for all the motivation and technical discussions that helped me gain a wider perspective of Science.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1
INTRODUCTION

Maximizing the performance while achieving a high degree of *energy efficiency* has become the central focus of microelectronic system design in practically every market segment - from battery powered mobile devices to high performance servers. Accelerators are a promising approach to improve the performance and power-efficiency of all such systems. At one extreme are special purpose, custom hardware accelerators. These have been shown to achieve the highest performance with the least power consumption. However, they are not programmable and incur a high design cost. At the other end of the spectrum are Graphics Processing Units (GPUs), which have become very popular. Although GPUs are programmable, they are limited to accelerating only *parallel loops*. In between these two extremes, are Field Programmable Gate Arrays (FPGAs). They have some of the advantages of hardware accelerators, and are also programmable. However, their *fine-grain* reconfigurability incurs a very high cost in terms of power and energy efficiency.

Coarse-Grained Reconfigurable Architectures or CGRAs have been shown to be an excellent alternative as they not only have power efficiencies close to hardware accelerators, but can be utilized for a wide range of applications because they are programmable. For instance, ADRES CGRA has been shown to achieve performance and power efficiency of 60 GOPS/W in 90 nm CMOS technology [3].

A CGRA is a collection of Processing Elements(PEs) connected through a mesh network, with each PE equipped with an ALU and a small register file (see Figure 1). The PEs are connected to their neighboring PEs, and the output of a PE is accessible to its neighbors. In addition, a common data bus from the data memory provides data to all the PEs in a row. It is referred to as coarse-grained reconfigurable because PEs can be programmed to execute different instructions at cycle level granularity.

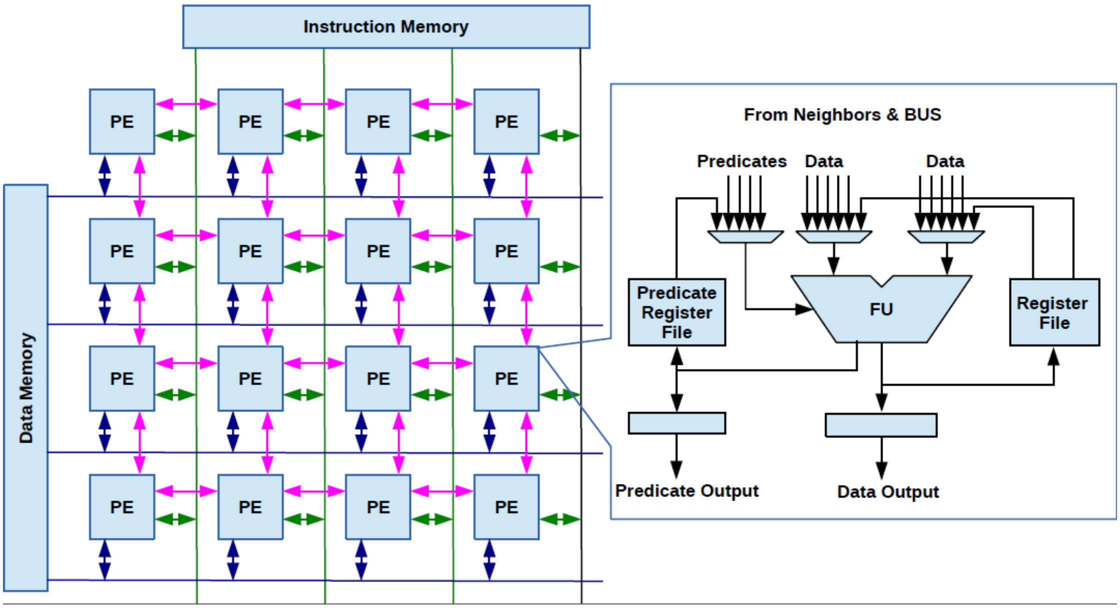


Figure 1: A 4 x 4 CGRA. A PE consists of an ALU and two register files, a data register file to hold data and a predicate register file. Predicate register file is used to execute instruction conditionally in the presence of control divergence in the code.

Applications execute in phases and often just a few phases or regions contribute most to the execution time. Those regions are usually composed of *loops*, and it is the acceleration of such loops that can significantly reduce the application execution time. Note that GPUs can accelerate such loops only if they have no dependencies across iterations. Acceleration of loops, even with dependencies across iterations, can be performed very efficiently using CGRAs. This is done by using a classical technique referred to as *software pipelining* [16], which reorders instructions. *Modulo scheduling* [21] is a form of software pipelining that allows overlapping the execution of successive iterations of a loop. This requires the use of a special register file, referred to as a *rotating* register file [22], which prevents a register index from being overwritten in successive iterations before that register is read and consumed by dependent operations.

A rotating register file is not sufficient on its own. This is because some operations require a register index that does not change during the execution. Such operations include loads and stores, and operations with constant operands. Thus both a rotating and non-rotating register files are needed, and this poses a unique problem for CGRAs. For instance registers that hold constant values would be difficult to index if the register indices change dynamically, which would be the case with a rotating register file.

In this thesis, we investigate different register file structures for CGRAs that can efficiently handle memory operations as well as short lived values. This problem is important because it is necessary to efficiently perform load and store operations in most computation segments. To this end, we present three different register file structures for CGRAs:

1. A programmable register(PRF) file at each PE which can be logically partitioned into rotating and non-rotating regions. In this case, the compiler must determine the boundary between rotating and non-rotating region for each PE's register file. This boundary is set for each at configuration time. Our experiments show that a programmable solution is the best in terms of performance and area. This structure enables us to accelerate a wider spectrum of applications and deliver better performance compared to the other solutions.
2. A rotating register file in each processing element (PE) and a shared non-rotating register file for the set of PEs in each row.
3. A fixed size rotating and non-rotating register file at each PE.

Chapter 2

MOTIVATION FOR ROTATING AND NON ROTATING RF

In this section we present the motivation behind having a rotating and non-Rotating register file within each PE. We first give a brief overview of modulo scheduling.

Figure 2 shows an illustration of how *Modulo scheduling* [21] helps in accelerating the execution of loops and makes CGRAs an excellent choice for the same. The performance metric of modulo scheduling is *initiation interval* or II, which is the required time between the initiation of two successive iterations of the loop. The II is inversely proportional to execution time.

The vertices of Data Flow Graph(DFG) represent the operations inside a nested loop and the edges represent the data dependencies between them. Figure 2(d) shows a valid mapping of the operations from the DFG of Figure 2(a) onto the CGRA of Figure 2 (b). As we can see each iteration requires 4 cycles to complete and the second iteration cannot begin its execution before the completion of the first iteration. Figure 2 (e) shows a modulo schedule generated mapping for the same DFG on the same CGRA. The execution of multiple iterations of the DFG using the modulo scheduled mapping is shown in Figure 2 (f). We can see that after exploiting the possibility of a software pipelined execution, we can initiate a new iteration every 2 cycles (II=2) and hence achieve a performance gain of 2X. The dotted region in Figure 2(f) comprises of all the operations from the loop body.

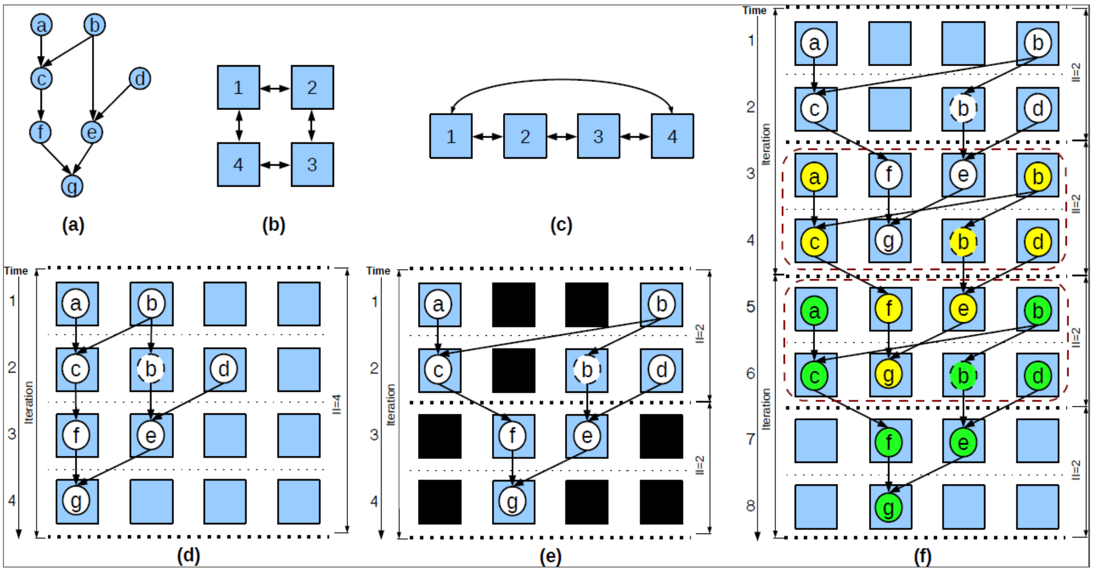


Figure 2: (a) an input DFG. The vertices indicates operations and edges indicate the data dependencies (b) A 2 x 2 CGRA (c) The CGRA in *b* shown horizontally (d) A valid mapping of the operations from DFG *a* onto the CGRA *b* (d) A modulo scheduled mapping of the operations from DFG *a* onto the CGRA *b* with an Initiation Interval of 2 (e) Actual execution sequence of the mapping in *d*

The steady-state of the modulo scheduling is usually referred to as a *kernel*. A kernel consists of an instruction for each PE for *II* cycles. Those instructions are repeatedly executed until the execution of a loop is completed. Note that the same kernel body, and therefore the same set of instructions, are executed every *II* cycles. Note that the register index encoded in an instruction cannot change (because it is the same instruction) across iterations. However, because the same instruction is executed in different iterations, and the destination register index does not change, it may overwrite a previous value in that destination register. This can cause a problem if another instruction requires that previous value in a future iteration. This is well known problem in VLIW processors, which has been addressed by the use of rotating register files [22].

In a rotating register file, the register indices are changed either logically or physically at the end of each loop iteration. In the logical approach, an offset to the register index is incremented at the end of each iteration. On the other hand, a physical

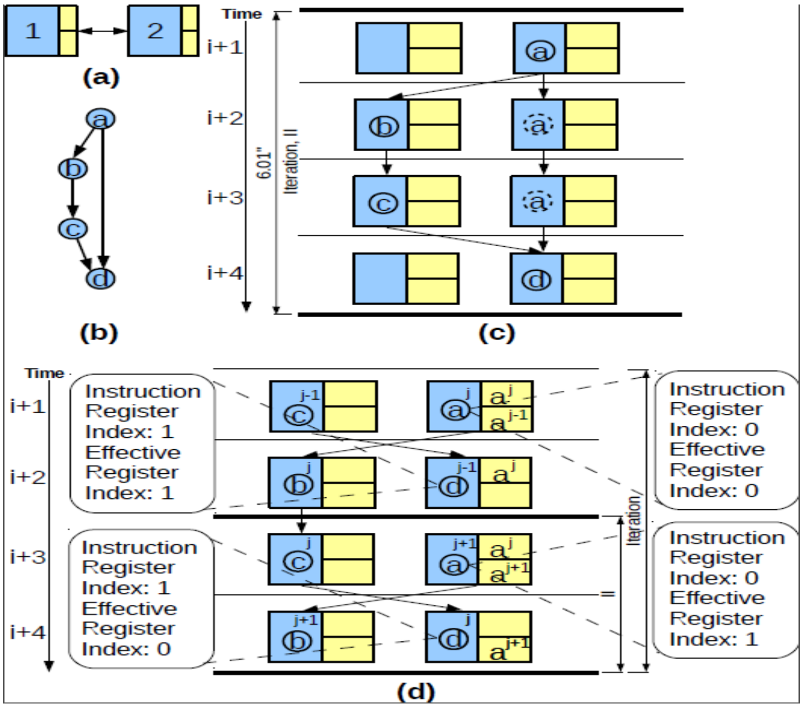

1
2
(a)
Time
i+1
i+2
i+3
i+4
a
b
c
d
(b)
6.01"
Iteration, II
(c)
Time
i+1
i+2
i+3
i+4
Instruction Register Index: 1 Effective Register Index: 1
Instruction Register Index: 1 Effective Register Index: 0
Instruction Register Index: 0 Effective Register Index: 0
Instruction Register Index: 0 Effective Register Index: 1
j-1
j
j+1
Iteration
II
(d)

Figure 3: (a) A 2 x 1 CGRA, (b) an input DFG, (c) a valid mapping of the given DFG (b) on (a) without using registers. The value of operation *a* is routed to the operation *d* through PEs. This mapping achieves initiation interval II = 4. (d) Another mapping, that uses registers to transfer value to operation *a* to operation *d*. This mapping achieves initiation interval II = 2. Lower II is achieved because two iterations of the loops are executed simultaneously which becomes possible because internal registers of *PE_2* are used to route data from *PE_2* at cycle 1 to *PE_2* at cycle 4.

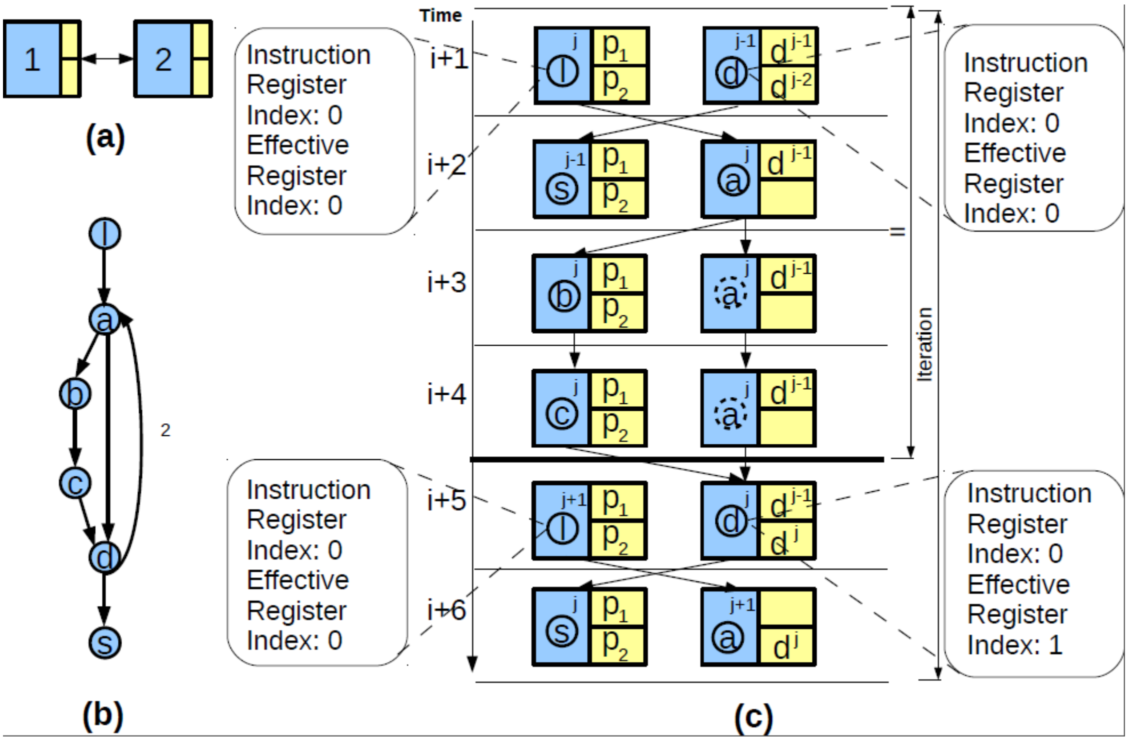
change requires the use of a shift register, that rotates the values in the register file. In either case, a value stored in the previous iteration at any index would not be overwritten in subsequent iteration when the same register index is selected as a destination. A rotating register enables a compiler to generate very compact code. Figure 3 shows how rotating register files enable an efficient loop execution.

Consider a 2 x 1 CGRA where each PE has 2 local registers, as shown in figure 3 (a). We intend to accelerate a loop whose data flow graph (DFG) has 4 operations, as depicted in figure 3(b). The first mapping shown in figure 3(c) requires 4 cycles to execute one iteration of the loop. The next iteration can be initiated after 4 cycles. Thus II of this mapping is 4. The second mapping shown in Figure 3(d) improves the performance by 2*X* because every 2 cycles, a new iteration of the loop can be initiated (*II=2*). The iteration label is shown as superscript. Iteration *j* starts at cycle *i+1* when a^j is executed on PE_2. The result of a^j is stored on register 0 of PE_2. At the next cycle, b^j is executed on PE_1. At cycle *i+3*, c^j is performed on PE_1. Finally, PE_2 executes d^j at cycle *i+4*. The DFG indicates that d^j requires the value of a^j stored in register 0 in PE_2. Therefore, following this schedule, it takes 4 cycles to completely execute one iteration.

Since all resources to execute the next iteration of the loop at cycle *i+3* and *i+4* are available (not used by operations that belong to iteration *j*), the next iteration can be initiated well before the previous iteration is completed. Specifically, the next iteration can be initiated at cycle *i+3* when PE_2 executes a^{j+1}. This results in a reduction in the *II* by a factor of 2, implying that the performance is increased by 2*X*.

his reduction in *II* is only possible when PEs are equipped with rotating register files. The steady-state of the pipeline is shown in figure 3(d) between the two thick lines from cycle *i+3* to *i+4*. Note that since *II=2,* the same instructions shown in cycle *i+3* and *i+4* are executed repeatedly. In steady state, the operations a^{j+1} and c^j are executed at cycle 0 (like *i+3*), and b^{j+1} and d^j at cycle 1 (like *i+4*).

Since the same instruction is executed at every iteration of steady state, the same register index is used to select the destination register index (index 0 or r_0). However, due to rotation of the register file (or register index), the effective destination index changes, which results in writing to a different register index. Figure 3(d) shows the usage of rotating register files by operations a and d in the DFG in 3(b) for an execution instance. The effective register index after the rotation is calculated as *((time + iteration number) % II)*. We can see that instruction a^j at cycle *i+1* and a^{j+1} at cycle *i+3* use the same destination index register (0), however, two different register indices (index 0 and 1) are updated when they are executed. This is an important feature because if a^{j+1} were to update the same index, a^j would have been overwritten at cycle *i+3*. Thus, at cycle *i+4*, result of a^j would have not been available to execute d^j. This rotation feature allows us to generate a very compact mapping. If not, the register indices have to be manipulated on every iteration, which requires execution of more instructions, and also increasing the *II*.



1
2
(a)
(b)
(c)
Time
i+1
i+2
i+3
i+4
i+5
i+6
Iteration
II
2
Instruction Register Index: 0 Effective Register Index: 0
Instruction Register Index: 0 Effective Register Index: 0
Instruction Register Index: 0 Effective Register Index: 0
Instruction Register Index: 0 Effective Register Index: 1

Figure 4: (a) a 2 x 1 CGRA, (b) an input DFG where *l* **is a load instruction from address** p_1 **and** *s* **is a store operation to address** p_2**. The arc between operation** *d* **and** *a* **has a weight of** *2* **representing dependency between operation** *a* **at iteration** *j* **and operation** *d* **at iteration** *j-2* **(c) a valid mapping of the given DFG (b) on (a) with iteration** *II=4* **and latency of** *6* **cycles. The superscript is used to represent the iteration number of the operations.**

While a rotating register file is a perfect structure for satisfying data dependencies between producer and consumer instructions in a loop, it imposes difficulties when we use them to hold addresses and constant values. Consider a more realistic DFG shown in Figure 4(b) with load and store operations.

Node *l* is a load instruction, and p_1 is the address from where data is to be loaded. It is increased by 4 every time *l* is executed (in the loop, we are loading from an integer array and moving to the next element in array in next iteration). Node *s* is a store instruction and p_2 is the address where data is to be stored. p_2 is also increased by 4 every time *s* is executed. There is an arc from node *d* to node *a* with a weight of 2. This represents dependency between a^j and d^{j-2}.

In this example, 4 registers are required: 2 registers to hold p_1 and p_2 for load and store operations, 2 registers to satisfy data dependency between a^j and d^{j-2} (as there is an inter-iteration data dependency). A valid mapping of this DFG onto a 2 x 1 CGRA is shown in Figure 4(c).

For registers in PE_1, it is necessary to have a non-rotating register file. If there is a rotating register file in PE_1, we cannot keep both p_1 and p_2 in the registers of PE_1 because p_2 would be overwritten by operation *l*. Consider a PE with a rotating register file where p_1 and p_2 are stored in register 0 and 1 respectively at iteration *j*. In the next iteration, when *l* is executed to update p_1, p_1 would be written to register 1 which holds value of p_2 (the index is increased from previous iteration in rotating register file). Therefore, we loose the value of p_2 and the store operation would update a wrong location.

Meanwhile, it is necessary to have a rotating register file in PE_2. It is because when d^j is executing on PE_2, it should not overwrite the value of d^{j-1} in its register file. Therefore, every time *d* is executed, it should update a different register index than the one it updated in previous iteration (so the index should change every iteration). Thus we need both a rotating register file in PE_2 and a non-rotating register file in PE_1 for mapping in Figure 4(d). In the following section we present several designs of a register file for CGRAs, that allow both types of registers.

Chapter 3

REGISTER FILE ORGANIZATION FOR EFFICIENT LOOP EXECUTION

In this section, we present an efficient register file design, which we refer to as a *programmable register file*(PRF), and compare it against two other register file organizations:

1. A rotating register file per PE and a shared non-rotating register file per row.
2. A register file per PE that is physically partitioned into rotating and non-rotating regions.

The programmable register file can be logically partitioned into rotating and non-rotating regions at run-time. The boundary between those regions is determined by the compiler and is set at configuration time for each PE.

3.1 Design I: Programmable Register File (PRF)

The PRF structure is derived by modifying the design of a rotating register file, which will allow logical partitioning into rotating and non-rotating regions at run-time. To better understand the new design, we first discuss the structure of a rotating register file presented in [9].

As stated earlier, in a rotating register file, the input register index is added by an offset value as shown in Figure 5. The result of this operation drives the input port of register bank. The offset register is incremented at the end of every iteration of the loop, or every II cycles. Only $log_2(n)$ bits are required to index a register bank with *n* registers. Therefore, the bit width of both offset counter and adder in this structure is $log_2(n)$ bits. Note that an overflow from an addition simply results in a modulo operation. It is because the higher bits are not used to index register bank. Similarly, when the offset counter reaches the maximum value *n* (We assume the number of registers, *n* is a power of 2), in the next iteration, it will reset to zero.

The logical partition of the register file into rotating and non-rotating regions can be achieved by adding a simple finite state machine (register control or RC) that controls the offset counter and register bank ports as depicted in Figure 6. At configuration time, RC receives a threshold number, *T*. Let $(x\_\{w-1\}, x\_\{w-2\}, ..., x\_0)_2$ be the binary representation of *T*, where $w= log_2(n)$ and *n* is the number of registers in register bank.

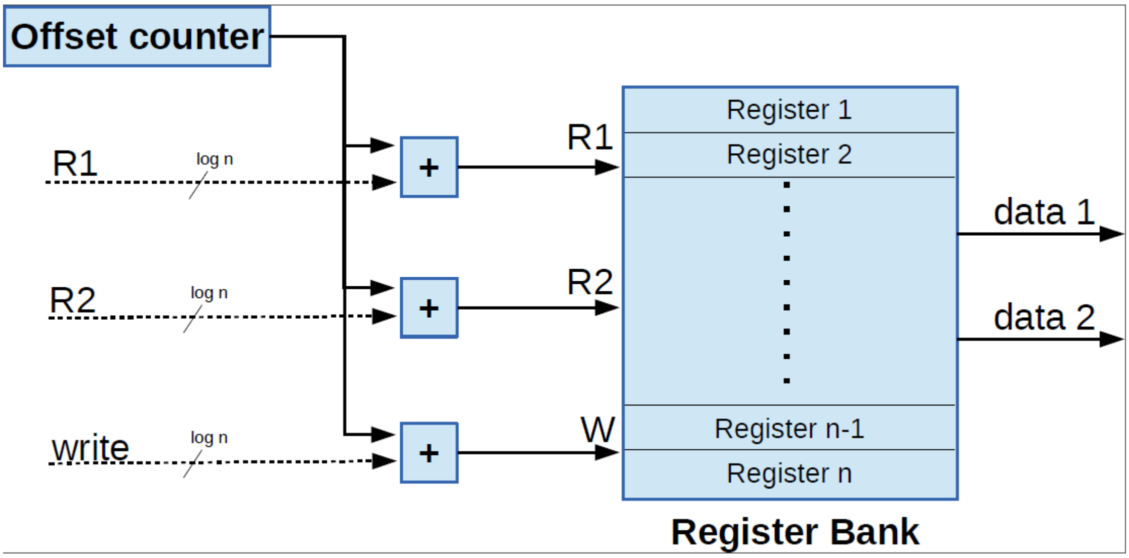


Figure 5: A rotating register file structure. Inputs to this register file is added to an offset value and then drive the register bank ports. This results in a logical rotation of register indexes in register file. The bit width of adder is designed to be equal to $log_2(n)$ **where** n **is the number registers in register bank.**

Since the register file is always on the critical path, it is important to minimize the delay overhead of any additional functionality. To this end, we limit the compiler to set T to values that can be represented as 2^i-1, 0 <= i <= $log_2(n)$. This limitation simplifies the path between the input register index and register bank port to form a modulo operation function. For a given i, all bits in T from position i to position $w-1$ are 0, while the rest of the bits are 1. Thus, if we perform a bitwise *AND* operation between T and output of the adder, it guarantees that the result is always less than T, while the lower bits of the index would not change. Thus, with a simple adder and a bitwise *AND* operation, we can implement the modulo operation (%) function, and emulate a rotation of the register indices.

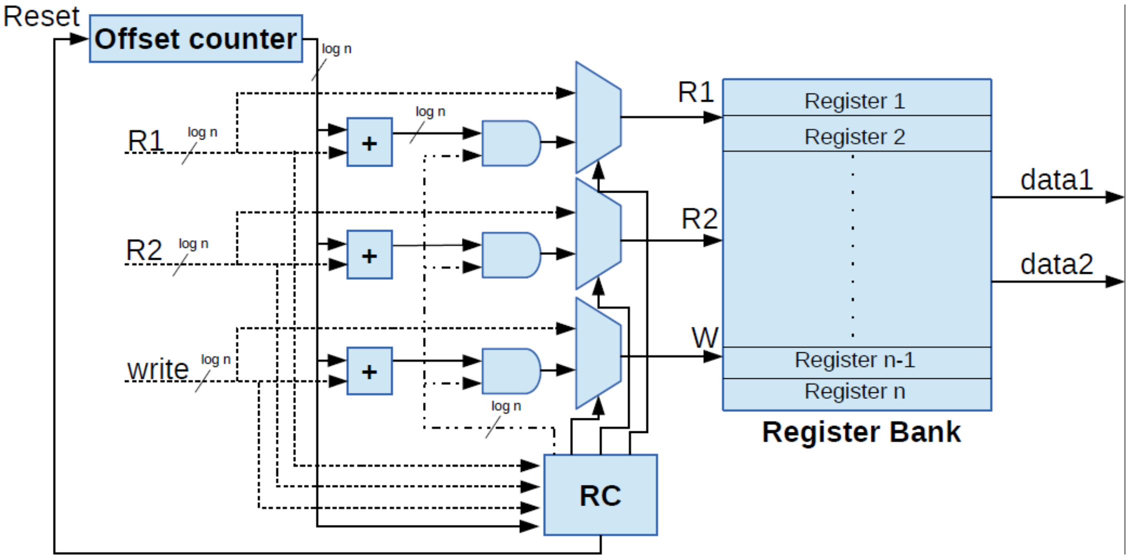


Figure 6: In this configuration, the register file can be logically partitioned into rotating and non rotating regions. This flexibility comes at the cost of extra components in RC unit.

An input register index of register file is sent to RC as well. If the register index is less than *T*, the output of bitwise *AND* operation drives the register bank port. Otherwise, the input register index is selected to drive the register bank. As shown in Figure 6, this structure imposes a minimal amount of logic overhead to the register file as compared to the design of the original rotating register file. Note that bitwise *AND* is a fully parallel operation.

At the end of an iteration, RC increments the offset counter and compares it against *T*. If the value of offset counter is greater than *T*, RC resets it to zero. Note that in this structure, RC explicitly resets offset counter register which is in contrast to the previous design where the value of offset register is implicitly reset to zero when it reaches the maximum value it can represent.

This simple change significantly increases CGRA reconfigurability. In this design, the border between rotating and non-rotating regions in a register file can be dynamically changed. Therefore, a compiler can allocate rotating and non-rotating registers in a flexible manner at each PE.

The proposed structure can also significantly simplify register allocation in the compiler. A compiler can map operations just based on the total number of registers needed at each PE instead of allocating rotating and non-rotating register separately. In contrast, existing CGRA compilers such as [7] have to keep track of the number of rotating and non-rotating registers separately because they have fixed the size of rotating and non-rotating register files.

The second benefit is that a wide spectrum of applications can be efficiently mapped by this structure. Some applications, such as those with high data dependencies between operations, impose a high demand on rotating register files, while other applications that have many load and store operations, place a high demand for non-rotating registers. As long as the total number of registers are sufficient in those applications, a PRF can effectively accelerate those loops. By fixing the number of rotating and non-rotating registers at design time, only a limited set of those applications can be effectively accelerated on CGRA. A PRF does not impose any change in instruction size. It, however, requires an increase in CGRA configuration size (only one instruction) to set RC thresholds.

3.2 Design II: A Rotating Register File per PE, a Shared Non-Rotating Register File per Row

We refer to this organization as a *Shared Non-Rotating Register File* (SNRRF). In this configuration, there is a rotating register file at each PE. In addition, there is a non-rotating register file at each row that is shared among all PEs in that row. On any cycle, only one PE in a row can update a register in a non-rotating register file. However, all PEs in a row can simultaneously read from this unit. This structure is shown in Figure 7.

A rotating register is usually used to temporarily hold an output of an operation that is to be used in next few iterations by one or more consumer operations. It is important to note that the number of registers in rotating register file has a direct impact in mapping II [12].

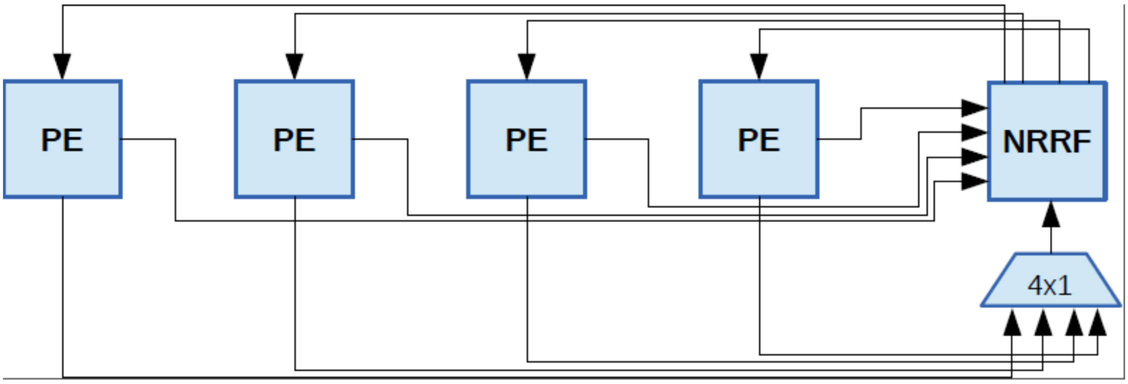


Figure 7: The shared register file organization. The output of all PEs in a row are sent to shared register file. However, only one PE per cycle can write to this register file. All PEs in a row can simultaneously read from shared register file.

Non-rotating register files are used to hold memory addresses and constant values that do not fit in the immediate field of instructions. Variables such as counters that are only alive within II cycles can also be kept in non-rotating register files because of the short schedule distance between producer and consumer. For instance, operation *l* updates the pointer address which is used by the same instruction at the next iteration in Figure 4. In fact, this instruction loads an element of a linear integer array. Assume that p_1 is initially pointing to the first element in that array. When it is executed, it increases the pointer by 4. Therefore, in next iteration, it would load the next element of the array. This is also the case for store operation *s* in figure 4(b). This is an example of a short distance between producer and consumer, thus, a non-rotating register file serves this dependency well.

The non-rotating register file structure can lead to an efficient register utilization. For instance, if a pointer is used in multiple load and store operations, we only need to allocate 1 register to hold that address if those instructions are mapped to PEs located at the same row.

The major problem with the non-rotating register file is that the number of rotating and non-rotating registers are fixed at design time. Therefore, it is not the total number of registers that determines whether or not an application can be accelerated well. Rather, either total number of rotating registers or total number of non-rotating registers can separately limit the CGRA to accelerate an application. Therefore, this structure cannot effectively accelerate a wide spectrum of applications.

Since a non-rotating register file is shared among PEs in a row, the shared register file should have a multi read/write port implementation. This imposes significant area overhead and degrades the CGRA design frequency. It is important to note that increasing the register file delay significantly impact the design frequency. In addition, only one PE per cycle can write to non-rotating register file, thus II might need to be increased to avoid write access conflicts between PEs. It may also increase the prolog length to initially store addresses and constants in non-rotating register files. Hence, there is a configuration time overhead associated with this design.

Last, this structure imposes overhead in the instruction size. Since non-rotating register file is shared amongst all PEs of a row, a separate field in an instruction bundle has to be dedicated to drive the write index of non-rotating register file. In addition, to be able to index shared registers as well as local registers, the register index field in an instruction has to increased to accommodate this need.

3.3 Design III: A Non-Rotating Register File per PE, and a Rotating Register File per PE

As shown in Figure 8, in this organization, each PE is equipped with a register file that is physically partitioned into rotating and non-rotating regions. We refer to this structure as *Fixed Register File* (FRF). Let's assume there are *n* rotating and *n* non-rotating registers at each PE. If a register index exceeds the rotating region limit (*index* > *n*), then that index simply bypasses the adder and drives the read port of register file. Thus, it would read from the non-rotating region. Along with offset counter, the adder is responsible to offset register index to act as a rotating register file. Note that the most significant bit of the add operation is statically assigned to zero.

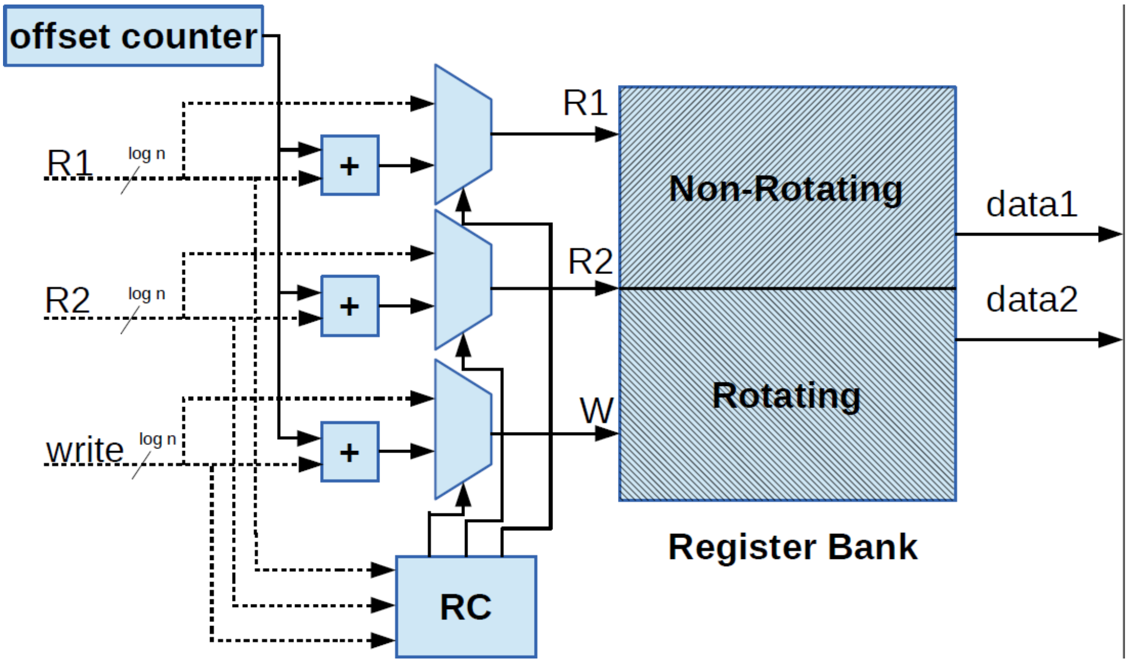


Figure 8: A register file that is physically partitioned into rotating and non-rotating regions.

The advantage of this structure over SNRRF (Design II) structure is that there is no need for a specific instruction to control the shared register file. In addition, there is less area and delay overhead because this register file in not shared. The delay and area of this structure is slightly better than PRF. It is because the bitwise *AND* operation is eliminated in this structure.

However, if multiple memory operations reading and writing from and to the same address are mapped onto different PEs, each PE has to allocate a register to keep the address separately.

Similar to SNRRF structure, this design cannot present an effective register utilization in wide spectrum of applications. It is because applications present varying need for rotating and non-rotating registers. Since the physical partition between rotating and non-rotating region is fixed and is set at design time, it cannot deliver an efficient resource availability if a wide spectrum of applications are to be executed on the CGRA. Also, the number of rotating and non rotating registers required is application dependent as well as this requirement changes from PE to PE.

Chapter 4
COMPILER SUPPORT

In this section we first present REGIMap [12], a register-aware CGRA mapping technique. We extend this technique to allocate both rotating and non-rotating registers. REGIMap initially extracts the minimum *II* using technique in [14]. Then operations are scheduled to minimize *II*. In the next step, a time extended resource graph is constructed. Afterwards, a compatibility graph *P* is generated from scheduled DFG and R_{II}. When *P* is constructed, a maximum clique $C=(V_C,E_C)$ in graph *P* where the sum of weight of outgoing arcs at all nodes is less than the register file size must be found. The mapping is completed when $|V_C|=|V_D|$. If REGIMap fails to find such a clique, it reschedules operations not present in the clique and tries again until a mapping is found. The REGIMap algorithm is presented in Algorithm 1.

Algorithm 1: REGIMap(Input D,Input CGRA)

```
begin
    $MII \leftarrow$ DetemineMII(D,$|V_D|$)
    $S \leftarrow |V_C|$; $D_S \leftarrow D$;
    while $true$ do
        $N \leftarrow \infty$
        while $true$ do
        $D_S, II \leftarrow$ Schedule($D_S, S$);
        if $II > MII$ then
            $MII \leftarrow MII + 1$;
            $S \leftarrow |V_C|$; $D_S \leftarrow D$;
            break;
        $R_{II} \leftarrow$ Construct_Resource_Graph($C$, $MII$);
        $P \leftarrow$ Construct_Compatibility_Graph($D_S$, $R_{II}$ );
        $C \leftarrow$ Weight_Constrained_Max_Clique($P$);
        if $|V_C| = |V_{Ds}|$ then
```

```
16.     return $C$;
17.     **else**
18.         **if** $|V_{Ds}| - |V_C| > N$ **then**
19.             $S \leftarrow S - 1$; $D_S \leftarrow D$;
20.             break;
21.             **else**
22.                 $D_S \leftarrow$ Re-Schedule($V_{Ds} - V_C$);
23.                 $N \leftarrow |V_{Ds} - V_C|$;
```

In this algorithm, the placement and register allocation is reduced to finding a clique in compatibility graph. The sum of arc weights in this graph represent the number of required registers per PE. Thus, if a node is selected to be added to the clique, the sum of arcs for this node is verified to be less than the number of available registers in a PE. We extend Weight_Constrained_Max_Clique(P) (line 14) function to verify the number of available registers when an operation is to be mapped on a PE. It should be noted that each node $i=(o_i, r_i)$ in graph P, represents a pair of an operation o_i and a PE r_i in resource graph.

Let C be the clique graph that is formed during mapping and i be a candidate node to be added to this clique. Shown in Algorithm 2, the number of total registers, rotating and non-rotating, is checked for PRF structure. In this algorithm, $R(o_i)$ returns the number of non-rotating registers required to map operation o_i. For instance, for a memory operation, a non-rotating register is required to hold the pointer address. If it is an instruction with a constant operand that is greater than 2^{15}-1 (in our CGRA ISA, only 16 bits are dedicated to immediate field which can be negative or positive), that operand is kept in non-rotating register.

A table is formed to keep track of the total number of allocated non-rotating registers for SNNRF design. In Algorithm 3, first the row index of a resource (PE) is found. Using this index, we first verify whether mapping of operation o_i increases the number of required non-rotating register beyond the number of available non-rotating registers per PE. If there are sufficient non-rotating registers available, this function verifies the number of rotating registers available at each PE. If it passes both of these conditions, the mapping of operation o_i on resource r_i is accepted.

Algorithm 4 depicts how the number of available rotating and non-rotating registers at each PE are checked for FRF design. In this function, the number of non-rotating registers required to map operation o_i on resource r_i is determined. We keep track of the number of allocated rotating and non-rotating registers separately. For each node *j* in the clique, if that node represents the same PE as r_i is representing, we increase the number of allocated non-rotating registers by non-rotating registers required by operation *j* is representing. We also keep track of rotating registers separately. In the end, we check if such mapping does not increase the number of rotating and non-rotating registers beyond what is available at each PE, the mapping represented by node *i* is accepted.

Algorithm 2: Can_Insert_PRF(Input $i = (o_i; r_i)$, Input Clique $C = (V_C;E_C)$, Input Register Size N)

```
begin
    $S \leftarrow 0$;
    for $\forall j \in V_C$ do
        if $e_{(i,j)} \notin E_C$ then
            return false;
        $S \leftarrow S + w_{(i,j)};$
        $S \leftarrow S + \mathrm{R}(o_i)$;
    if $S > N$ then
        return false;
    return false;
```

Algorithm 3: Can_Insert_SRF(Input $i = (o_i; r_i)$, Input Clique $C = (V_C;E_C)$, Input PE Register Size N, Input SRF Size M)

```
begin
    if $Table[get\_row(r_i)] + R(o_i) > M$ then
        return false;
    $S \leftarrow 0$;
    for $\forall j \in V_C$ do
        if $e_{(i,j)} \notin E_C$ then
```

```
                return false;
        $S \leftarrow S + w_{(i,j)};$
    if $S > N$ then
        return false;
    return false;
```

Algorithm 4: Can_Insert_FRF(Input $i = (o_i; r_i)$, Input Clique $C = (V_C;E_C)$, Input NRF Size N, Input RRF Size M)

```
begin
    $S_r \leftarrow R(o_i);$
    $S_n \leftarrow 0;$
    for $\forall j = (o_j,r_j) \in V_C$ do
        if $e_{(i,j)} \notin E_C$ then
            return false;
        if $PE(r_j) == PE(r_i)$ then
            $S_n \leftarrow S_n + R(o_j);$
        $S_r \leftarrow S_r + w_{(i,j)};$
    if $S_r > N$ then
        return false;
    if $S_n > M$ then
        return false;
    return false;
```

Chapter 5
EXPERIMENTAL RESULTS

The CGRA with different register file configurations was specified in RTL to evaluate the overhead associated with each register file structure. The various configurations were synthesized using Cadence RTL Compiler using a CMOS 65nm TSMC technology.

The REGIMap [12] algorithm is the base mapping technique used to support all of these register file structures. It is then integrated as a separate pass in the llvm compiler framework [6]. We also modeled CGRA as an accelerator in the GEM5 system simulation framework [2]. Loops that are important for performance were selected using Livermore Compiler Analysis Loop Suite [1] benchmark. Those loops represent typical nested loops in scientific codes. Experiments were conducted to evaluate the advantages and disadvantages of each structure in benchmarks.

The loops were mapped on to a 4 x 4 CGRA with sufficient instruction memory to hold all instructions within a loop body, as well as sufficient data memory space to hold all the variables. Latency of all the operations were assumed to be only one cycle. Load and store operations requires two CGRA operations, one for the address bus transaction and the other for the data bus transaction. The address and data buses are shared among all PEs within a row. In other words, only one memory transaction can proceed at any cycle in a row. We conducted experiments on mesh-interconnected CGRA.

As we stated earlier, we need non-rotating registers to perform memory operations efficiently. Hence, in our setup, the number of memory operations that can be performed in a loop kernel is limited by the total number of non-rotating registers present in the CGRA. Once the total number of registers is fixed, the proposed PRF design will be able to perform the maximum number of memory operations as all the registers in it can be configured to behave as non-rotating structures.

5.1 PRF configuration maps loops with minimum number of registers

In our first experiment, we change REGIMap [12] to increase the number of available registers for each configuration until the first mapping can be found (starts from 0). The results can be seen in Figure 9 which shows the minimum number of registers required by each register configuration to find a valid mapping. This is by far the most important factor to evaluate the effectiveness of each structure. It shows that a mapping can be found with the minimum number of registers when we use PRF in CGRA. Shared register structure requires relatively more registers even though it enables the maximum register sharing possibility for PEs.

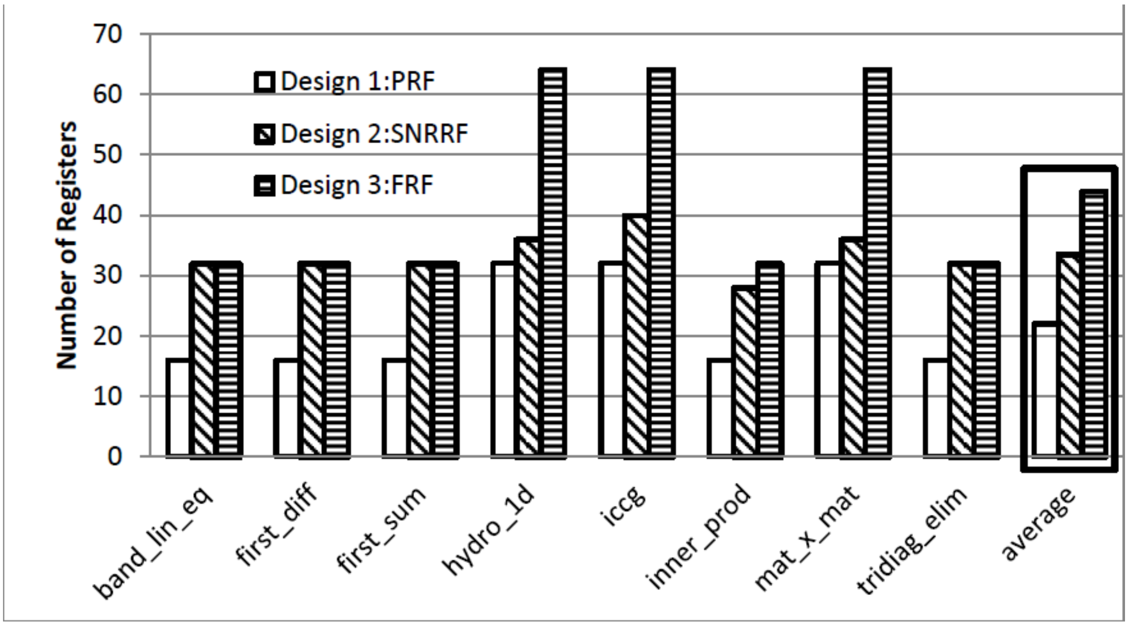


Figure 9: The minimum number of registers required for each RF configuration to find a mapping.

This is an important factor because it proves that with a given number of registers, can enable us to accelerate significantly more applications. This is crucial to a programmable accelerator such as CGRA because it is designed to be used as a general purpose accelerator rather than specialized accelerator.

5.2 PRF configuration imposes a minimal area overhead

Figure 10 shows the synthesis results for the three proposed register file structures. For a fair comparison, we have configured the CGRAs with a total of 64 registers. Fixed RF and

Shared RF have equal number of rotating and non-rotating registers (32 rotating and 32 non-rotating in each structure). PRF has a total of 64 registers i.e. 4 registers per PE.

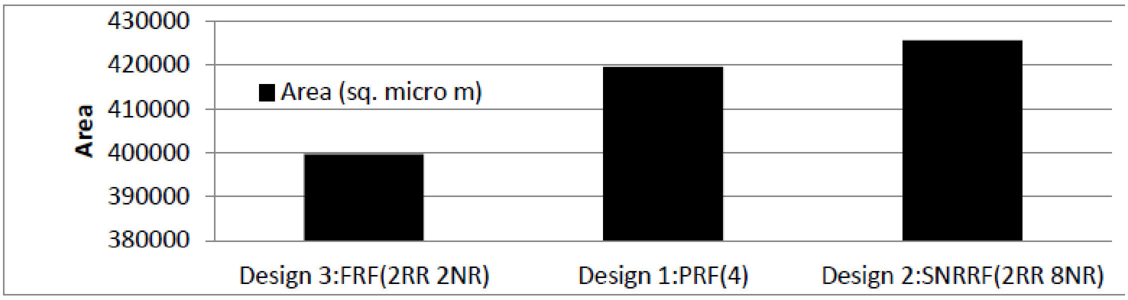


Figure 10: The area overhead imposed by each RF configuration. The PRF configuration imposes less area overhead compared to shared RF configuration.

CGRA synthesized with Fixed RF has the least area as all the PEs have their own Register Files without any shared registers amongst the PEs. Also, the boundary between the rotating and non-rotating region is fixed at the design time.

Even though the PEs of the CGRA using PRF structure do not share any registers, its area is slightly more than that of Fixed RF configuration because of the area overhead imposed by extra hardware required to dynamically configure the boundary between the rotating and non-rotating regions. To enable this feature an extra register (The size of this register is logarithmically proportional to the size of register file} is added to each PE and that accounts for the increase in area.

CGRA synthesized with Shared RF has the highest area overhead because all the PEs in a row share the non-rotating registers. This imposes extra multiplexer at the input and output of this register file. This leads to an increase in the number of ports and hence the area. We can see that the differences in the area of all the three structures is negligible and lies in the error range of the synthesis tool (Because synthesis tools use many approximations and non-deterministic algorithms for area and frequency estimation).

5.3 PRF imposes a minimal frequency overhead

Figure 11 shows the synthesis results for the frequency of the CGRAs synthesized with the same configuration as above. CGRA synthesized with Fixed RF structure has the highest frequency because of the design regularity. However, it provides the least flexibility in terms of register file usage.

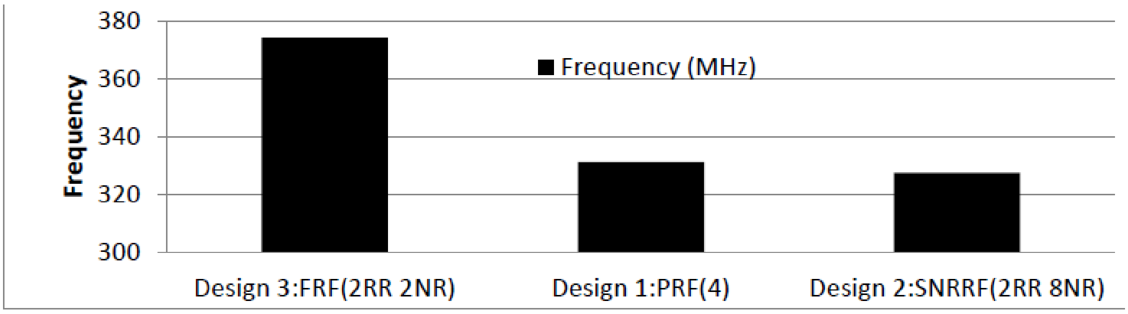


Figure 11: The Fixed RF configuration results in the best frequency. PRF results in slightly better frequency than shared RF structure.

The PRF structure has a slightly lower frequency as compared to the Fixed RF. But this register file structure provides a lot of flexibility in terms of register file usage which in turn leads to a very efficient mapping with a lower II and hence shorter schedules. The Shared RF structure has the lowest frequency amongst the presented register structures. This can be accounted by the fact that all the PEs in a row have access to a shared RF and this leads to slow register reads/writes.

5.4 SNRRF and PRF required close number of registers for a given II

In our next experiment, we fix II but increase the number of registers (starting from 0) in all configurations until a mapping at that II is found. This provides a fair comparison between these configurations to show which one can utilize registers in a better way.

Note that shared structure is the only configuration which enables register sharing between PEs. Thus we expect this structure to require least number of registers. In addition, for fixed and PRF, the total number of registers are increased by a factor of 16 (total number of PEs or in other words one more reg per PE). This is not the case for shared structure. The number of registers in this structure can increase by a factor of 4 (there are 4 rows). Thus, registers are increased in a finer granularity for this structure.

As can be seen in Figure 12, the total number of required registers in shared and PRF configuration are in fact very competitive. We conclude that the flexibility of partitioning the register file in PRF structure compensates its limitation on sharing registers very well.

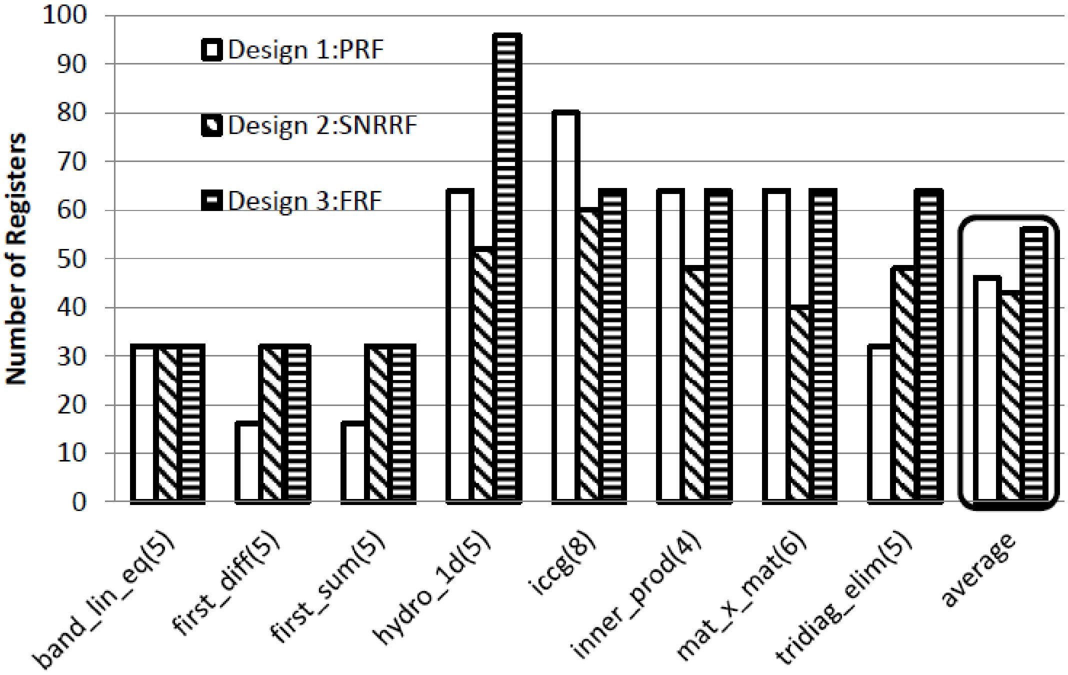


Figure 12: Total number of registers to achieve an II. On average, the number of registers required in PRF and Shared RF configurations is relatively close. The lower number of Shared is because the total number of registers in this configuration is increased in finer granularity.

Even though Fixed structure is very similar to PRF, it results in low register utilization. On an average, to achieve the same performance, 43,46,56 registers are required in shared, PRF, and fixed register configurations respectively. Note that the better results of shared structure is also because of finer increase in number of registers.

5.5 Mapping limitations of SNRRF

In this experiment, we keep the total number of registers in the CGRA constant and vary the number of rotating registers per PE and the number of non rotating registers in the shared register file at each row. The results are shown in Table 1. Observing this table, we can see that the Shared RF structure cannot be used for a wide spectrum of applications because we need to decide on one configuration at the design time. Hence, it is possible that the one configuration of Shared RF structure accelerates an application very well whereas results in No Mapping(NM) or extremely inefficient acceleration in others.

Table 1: The effect of changing the ratio of rotating and non-rotating registers on II

Kernel	Total Registers	NR/Row	R/PE	II
band_lin_eq	32	8	0	9
band_lin_eq	32	4	1	5
band_lin_eq	32	0	2	NM
first_diff	32	8	0	5
first_diff	32	4	1	5
first_diff	32	0	2	NM
first_sum	32	8	0	5
first_sum	32	4	1	5
first_sum	32	0	2	NM
hydro_id	52	13	0	7
hydro_id	52	9	1	7
hydro_id	52	5	2	5
hydro_id	52	1	3	NM
iccg	60	15	0	NM
iccg	60	11	1	9
iccg	60	7	2	8
iccg	60	3	3	NM
inner_prod	48	12	0	5
inner_prod	48	8	1	5
inner_prod	48	4	2	4
inner_prod	48	0	3	NM
mat_x_mat	40	10	0	7

mat_x_mat	40	6	1	6
mat_x_mat	40	2	2	NM
tridiag_elim	48	12	0	6
tridiag_elim	48	8	1	6
tridiag_elim	48	4	2	5
tridiag_elim	48	0	3	NM

5.6 SNRRF and FRF organizations are restrictive

There is an important problem that designers have to address if they choose to use shared structure: how many registers are to be a part of the NRRF and how many are to be a part of the RRF? This problem is visible in Table 1. For this experiment, we fix the total number of registers in CGRA, but vary the ratio between the number of registers in RRF and the number of registers in the shared register structures. This has an important effect on CGRA performance and the spectrum of applications it can actually accelerate. There are applications where the number of memory operations are small. However, there is heavy data dependency between operations. For such applications, the best performance can be achieved when we assign more registers to RRF.

For instance in *hydro_1d*, the performance significantly increases when RRF increase to 2 (per PE). However, further increasing of it will lead to failure in finding any mapping. On the other hand, applications such as *first_diff* do not benefit from more rotating register. This is an important burden for CGRA to be used as a general purpose accelerator.

Chapter 6

RELATED WORK

During the past decade, there has been an extensive research on CGRA designs. This resulted in many inspiring architectures including ADRES CGRA [3], PADDI [4], PipeRench [10], KressArray [13], Morphosys [17], MATRIX [18], and REMARC [19]. Although rotating register files have beend extensively investigated in CGRAs, non-rotating register files have been just overlooked.

Recently, there has been a shift to developing automatic mapping techniques to effectively utilize CGRAs. It is because without an efficient compiler, it is impractical to use CGRAs in real applications. Those inspiring mapping techniques include [5,20,11,12]. Several mapping techniques address register allocation along with mapping. This includes REGIMap [12] and [7]. These algorithms extensively investigate the allocation of registers in rotating register files present at each PE. However, the problem of register allocation in non-rotating register files is missing from CGRA compiler literature.

Shared register file structures and their configurations have been investigated in [9,15]. They explored register file structures for long lived and short lived values from the hardware perspective. Here short lived and long lived values refer to data dependencies and constants respectively. Also, they present effective register file configurations in terms of degree of connectivity, the number of ports, and the number of registers in the RFs, and their respective performance in terms of Instructions Per Cycle (IPC).

Rotating register files when they are directly attached to PEs are studied in ADRES CGRA [3] and RaPiD [8]. Non-rotating register files are studied in [9] and [15]. They studies different configuration of rotating and non-rotating register files from shared one to local register files at PEs. However, those papers only address the register file from hardware perspective.

Chapter 7

CONCLUSIONS

In this work, we show the problems associated with loops containing memory operations and present three register file structures that solve those problems efficiently. We presented the advantages and disadvantages of all the three structures both in terms of the hardware overhead they impose and their effects on the mappings generated for loop nests. With our experiments we are able to show that a CGRA synthesized with PRF is the best register file structure for loop executions on CGRAs as it provides a lot of compiler flexibility with a minimal area overhead and a minimal decrease in frequency.

REFERENCES

[1] Lcals: Livermore compiler analysis loop suite, 2013.

[2] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2 (Aug. 2011), 1–7.

[3] BOUWENS, F., BEREKOVIC, M., SUTTER, B. D., AND GAYDADJIEV, G. Architecture enhancements for the adres coarse-grained reconfigurable array. In Proc. HiPEAC (2008), pp. 66–81.

[4] CHEN, D., AND RABAEY, J. A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths. Solid-State Circuits, IEEE Journal of 27, 12 (Dec 1992), 1895–1904.

[5] CHEN, L., AND MITRA, T. Graph minor approach for application mapping on cgras. In Field-Programmable Technology (FPT), 2012 International Conference on (Dec 2012), pp. 285–292.

[6] CHRIS LATTNER AND VIKRAM ADVE. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[7] DE SUTTER, B., COENE, P., VANDER AA, T., AND MEI, B. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (2008), LCTES ’08, pp. 151–160.

[8] EBELING, C., CRONQUIST, D., AND FRANKLIN, P. Rapid â˘A ˘T reconfigurable pipelined datapath. In Field-Programmable Logic Smart Applications, New Paradigms and Compilers, R. Hartenstein and M. Glesner, Eds., vol. 1142 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 126–135.

[9] ESSEN, B. V., PANDA, R., WOOD, A., EBELING, C., AND HAUCK, S. Managing short-lived and long-lived values in coarse-grained reconfigurable arrays. In FPL (2010), IEEE, pp. 380–387.

[10] GOLDSTEIN, S., SCHMIT, H., MOE, M., BUDIU, M., CADAMBI, S., TAYLOR, R., AND LAUFER, R. Piperench: a coprocessor for streaming multimedia acceleration. In Computer Architecture, Proceedings of the 26th International Symposium on (1999), pp. 28 –39.

[11] HAMZEH, M., SHRIVASTAVA, A., AND VRUDHULA, S. Epimap: using epimorphism to map applications on cgras. In Proceedings of the 49th Annual Design Automation Conference (2012), ACM, pp. 1284–1291.

[12] HAMZEH, M., SHRIVASTAVA, A., AND VRUDHULA, S. Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In Proc. DAC (2013), pp. 18:1–18:10.

[13] HARTENSTEIN, R., HERZ, M., HOFFMANN, T., AND NAGELDINGER, U. Using the kress-array for reconfigurable computing. In Proc. SPIE (1998), pp. 150–161.

[14] HUFF, R. A. Lifetime-sensitive modulo scheduling. In Proc. PLDI (1993), pp. 258–267.

[15] KWOK, Z., AND WILTON, S. J. E. Register file architecture optimization in a coarse-grained reconfigurable architecture. In Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on (April 2005), pp. 35–44.

[16] LAM, M. Software pipelining: an effective scheduling technique for vliw machines. In Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (New York, NY, USA, 1988), PLDI ’88, ACM, pp. 318–328.

[17] LEE, M.-H., SINGH, H., LU, G., BAGHERZADEH, N., KURDAHI, F. J., FILHO, E. M. C., AND ALVES, V. C. Design and implementation of the morphosys reconfigurable computing processor. J. VLSI Signal Process. Syst. 24 (2000), 147–164.

[18] MIRSKY, E., AND DEHON, A. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In Proc. FPGAs for Custom Computing Machines (1996), pp. 157 –166.

[19] MIYAMORI, T., AND OLUKOTUN, K. Remarc: Reconfigurable multimedia array coprocessor. IEICE Trans. on Information and Systems (1998), 389–397.

[20] PARK, H., FAN, K., MAHLKE, S. A., OH, T., KIM, H., AND KIM, H.-S. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (New York, NY, USA, 2008), PACT ’08, ACM, pp. 166–176.

[21] RAU, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. In Proc. MICRO (1994), pp. 63–74.

[22] RAU, B. R., LEE, M., TIRUMALAI, P. P., AND SCHLANSKER, M. S. Register allocation for software pipelined loops. In Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (1992), pp. 283–299.