REGIMap: Register-Aware Application Mapping on Coarse-Grained Reconfigurable Architectures (CGRAs)

Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ
{mahdi, aviral.shrivastava, vrudhula}@asu.edu

ABSTRACT

Coarse-Grained Reconfigurable Architectures (CGRAs) are an extremely attractive platform when both performance and power efficiency are paramount. Although the power-efficiency of CGRAs can be very high, their performance critically hinges upon the capabilities of the compiler. This is because a CGRA compiler has to perform explicit pipelining, scheduling, placement, and routing of operations. Existing CGRA compilers struggle with two main problems: 1) effectively utilizing the local register files in the PEs, and 2) high compilation times. This paper significantly improves the state-of-the-art in CGRA compilers by first creating a precise and general formulation of the problem of loop mapping on CGRAs, considering the local registers, and from the insights gained from the problem formulation, distilling an efficient and constructive heuristic solution. We show that the mapping problem, once characterized, can be reduced to the problem of finding maximal weighted clique in the product graph of the time-extended CGRA and the data dependence graph of the kernel. The heuristic we've developed results in average of 1.89 X better performance than the state-of-the-art methods when applied to several kernels from multimedia and SPEC2006 benchmarks. A unique feature of our heuristic is that it learns from failed attempts and constructively changes the schedule to achieve better mappings at lower compilation times.

1. INTRODUCTION

The *holy grail* of computer hardware and software design across all market segments, including battery powered hand-held devices, tablets and laptops, desktop PCs, and high performance servers, is to simultaneously improve performance and power-efficiency (performance-per-watt). Multicores solve this problem to some extent, but accelerators are needed to improve power-efficiency to levels much higher than what multicores can provide. Although special purpose or function specific hardware accelerators (e.g. for FFT) can be very power efficient, they are expensive, not programmable, and therefore limited in usage. Graphics Processing Units (GPUs) are becoming very popular; although programmable, they are limited to accelerating only "parallel loops." Field Programmable Gate Arrays are general-purpose, but due to their fine grain reconfigurability, have poor power-efficiency. In this field of accelerators, Coarse-Grain Reconfigurable Architectures or CGRAs are a very attractive platform [24]. A CGRA is simply a two dimensional mesh of PEs, with each PE equipped with an ALU and a small register file (Figure 1). The PEs are connected to neighboring PEs, and the output of a PE is accessible to its neighbors in the next cycle. 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 each PE can be programmed to execute different instructions at the cycle level granularity.

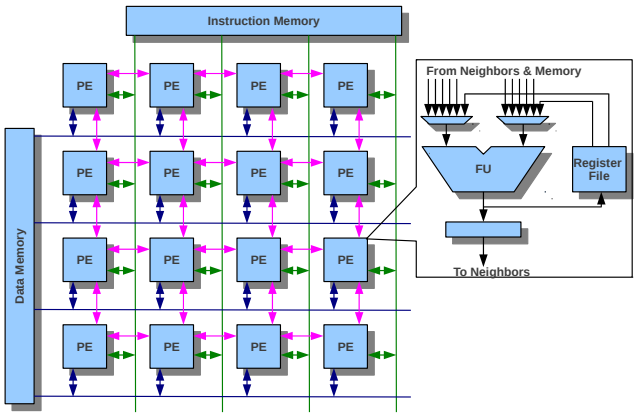


Figure 1: A 4×4 CGRA. PEs are connected in a 2-D mesh. Each PE is an ALU plus a local register file.

CGRAs are much more power-efficient, with power efficiencies close to hardware accelerators. They are also programmble and easier to program than FPGAs due to their coarse-level of reconfigurability. Finally, they are more general purpose accelerators than GPUs, as CGRAs can accelerate even non-parallel loops[1].

Attracted by the promise of CGRAs, more than a dozen CGRAs including XPP [2], PADDI [4], PipeRench [9], KressArray [13], Morphosys [18], MATRIX [20], and REMARC [21] were proposed over the past decade. In particular, the ADRES CGRA has been shown to achieve power efficiency of 60 GOps/W in 32 nm CMOS technology [3]. However, achieving the promised power-efficiency critically hinges on the compiler technology, and a CGRA compiler is much more complex than a regular compiler, since it has to perform code analysis to extract parallelism, schedule operations in time, bind the operations to PEs, route the data dependencies between the PEs, and perform explicit software pipelining. One major limitation of existing CGRA compilers is their inability to

[1]Non-parallel loops cannot be efficiently accelerated by GPUs; they can only be accelerated to a theoretical extent, depending on the inter-iteration loop dependencies. However, CGRAs permit acceleration of such loops.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC'13, May 29 - June 07, 2013, Austin, TX, USA.
Copyright 2013 ACM ACM 978-1-4503-2071-9/13/05 ...$15.00.

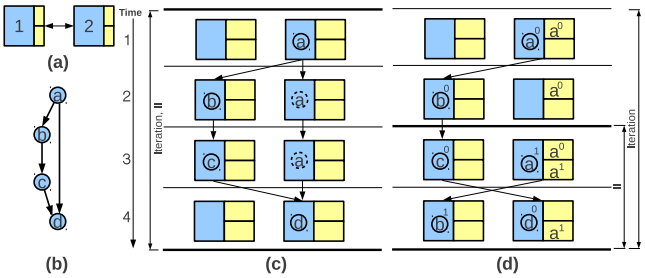


Figure 2: (a) a 2 × 1 CGRA, (b) an input DFG, (c) a valid mapping of the given DFG (b) on (a) with iteration latency =II= 4, (d) another mapping for the given DFG with iteration latency= 4 and $II = 2$. Lower II is achieved because two iterations of the loops are executed simultaneously which becomes possible because internal registers of PE_2 are used to route data from PE_2 at cycle 1 to PE_2 at cycle 4.

use register files to improve performance. Most existing compilers simply do not use register files, and transfer the operands among the PEs through a computational path with a PE. We know of only two schemes [6, 22] that generate mapping that use the registers in the CGRA. However, they suffer from the twin problems of i) poor acceleration of loops, and ii) high compilation times.

Towards improving the compiler technology for CGRAs, this paper makes several contributions:

1. A precise formulation for the CGRA mapping problem while using register files: We formally define the problem of register aware application mapping on CGRAs. In contrast to the previous ad-hoc problem definitions, our problem formulation is quite general and allows for various flexibilities in constructing mappings, including recomputation, and sharing of routing paths with dependencies.

2. A novel formulation for integrated operation placement and register allocation: We show that the problem of simultaneous placement and register allocation can be formulated as one of finding a *constrained maximal clique* on the product of two graphs: the data flow graph (DFG) of the computation and the graph representing a time-extended CGRA.

3. An effective heuristic for mapping loops to CGRAs: From the insights gained from the problem definition, we derive an effective heuristic to partition the mapping problem into sub-problems: (1) scheduling and (2) integrated placement, and register allocation. Our method, iteratively and constructively, solves these problems until a mapping can be found.

Experimental results on multimedia applications loops and some SPEC2006 benchmark kernels demonstrate the effectiveness of our heuristic. Our approach improves the performance of computationally bounded loops on average by 1.89 X, while reducing the compilation time by 56X than the existing state-of-the art technique [6].

2. BACKGROUND AND RELATED WORK

More than a dozen CGRA architectures have been designed till now [12]. These CGRAs were primarily targetted for embedded systems to perform signal processing in a power-efficient manner, and therefore programmed manually. However, as we envision the use CGRAs as more general purpose accelerators, compiler technology to automatically map parts of applications onto CGRAs is needed. Recognizing this, much of the research on CGRAs in this century has focused on advancing compilation for CGRAs [1, 7, 8, 11, 14, 17, 22, 25].

Since applications spend most of the time on loops [23], existing compilation techniques (as well as this paper) focus on the problem of mapping the innermost loops on a CGRA. Figure 2(a) shows a CGRA with 2 PEs, and Figure 2(b) shows the data flow graph of a simple loop. Figure 2(c) shows one valid mapping of the data flow graph on the CGRA. In this graph, the CGRA is extended in time to 4 cycles. In cycle 1, operation a is performed on PE_2. Operations b is executed on PE_1 in cycle 2, receiving the value of operation a from the output register of PE_2 (see Figure 1. Note each PE has an output register). Similarly operation c and d are executed on PE_1 PE_2 at times 3 and 4, respectively. To enable this computation, the result of operation a must remain in the output register for 3 cycles. This is also called *routing* of the dependency from a to d. The schedule length of this mapping is 4 cycles, and the II (Initiation Interval) of this mapping is also 4 cycles. II means the difference in time between the initiation of successive iterations of the loop. Since performance is inversely proportional to II, the goal of mapping is to minimize II (rather than schedule length).

Note that the mapping in figure 2(c) does not use registers in the PE. It routes the dependencies through PEs (actually the output register in the PE). As shown in the figure, each PE has 2 registers. Figure 2(d) shows the mapping using the registers in the PEs. In the mapping, after a is executed on PE_2 at cycle 1, its result is stored in one of its registers until d is executed by PE_2 at time 3. The result of a is also made available via its output register to PE_1, which executes b at cycle 2. Although the schedule length of this mapping is 4, the II is reduced to 2. This is because the next iteration of the loop can start at cycle 3. This is shown in the figure by operations a^1, b^1 being mapped in cycles 3 and 4. All operations are present in cycles 3 and 4, and this kernel can execute repeatedly, giving an II of 2.

This simple example shows how using register files can result in higher performance of loops on CGRAs, but most of the existing compiler techniques for CGRAs do not exploit register files well. We know of only two techniques that do use register files to obtain better mappings. However, both the existing techniques are "exploratory" mapping techniques. EMS [22] allocates registers during the scheduling and placement of operations on CGRA. The method places the input DFG, arc by arc, onto the CGRA. An arc can be placed, if the nodes on the arc can be placed, and the dependencies can be routed. If at any point this cannot be done, II is increased, and this mapping is tried all over again. The method described in [6], called DRESC, expands the time-extended CGRA graph to explicitly include registers as nodes (one node per register file). The method uses simulated annealing to find a mapping. Operations are randomly moved to decrease the number of overused resources. Once all resources are used only once, the mapping is completed. If no mapping can be found, II is increased and mapping is tried again. No control strategy, e.g. the temperature schedule is derived for the algorithm.

The drawbacks of the existing techniques, namely poor mapping and high compilation times are due to their exploratory nature. The basic strategy employed is to find two mappings successively: first, a mapping of the operations in the DFG to the PEs of the time-extended CGRA, followed by a mapping of arcs in the DFG to paths between the corresponding PEs in the time-extended CGRA. Even though the mappings allow paths to start and end in PEs, with the intermediate nodes being allowed to be PEs or registers, the approaches is restrictive as they do not permit recomputation, in which one operation of the DFG can be mapped to multiple PEs in the CGRA [11].

In this paper, we will present a general formulation of the problem of mapping a kernel on the CGRA while using its registers to minimize II. The formulation enables us to partition this problem into a scheduling problem and an integrated placement and register allocation problem. We reduce the placement to one of finding a *constrained maximal clique* in the product graph of the time-extended CGRA and the input DFG. Then we develop an efficient and constructive heuristic to map loop kernels onto the CGRA. An

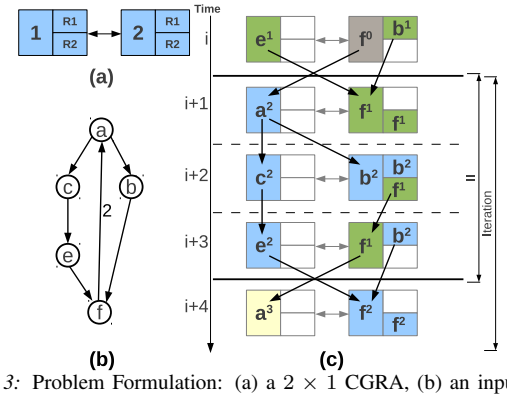


Figure 3: Problem Formulation: (a) a 2 × 1 CGRA, (b) an input DFG, (c) Mapping of the DFG to a CGRA. II of this mapping is 3 but to better visualize the overlap of loop execution, operations from iteration 1 and 3 are also shown.

important aspect of our heuristic is its ability to learn from failures, which in turn results in better performance, at lower compilation times.

3. PROBLEM DEFINITION

Let $I = (V_I, E_I)$ be the input DFG representing a loop. Given a DFG and a CGRA, we first determine the lower bound on II by extracting resource minimum II ($ResMII$) and recurrence minimum II ($RecMII$) [23], denoted as MII. Given an MII, we make a time extended resource graph denoted by $R_{II} = (V_R, E_R)$ which is constructed by replicating the nodes in the CGRA (PE and registers), II times, representing cycles 0 through $II-1$. For every pair (u, v) of adjacent nodes in the CGRA, there is an arc from (replication of) u at time t to (replication of) v at time $t+1$. Note that every node in the CGRA is adjacent to itself. This time-extended resource graph is the same as the MRRG graph used in [6].

Figure 3 (a) shows a 2 × 1 CGRA, and (b) shows input DFG. First the minimum II of this is calculated as 3. The CGRA graph is then unrolled 3 times, and a time-extended CGRA resource graph is constructed. We do not show that graph in the figure, since it has too many arcs, and does not help in visualization. Figure 3 (c) shows the mapping of the input DFG on this time-extended resource graph. The time-extended CGRA must be extended only 3 times (and therefore 3 rows), but we add an extra row at the top and bottom for clarity. The following terminology is needed to simplify the statement of the problem.

Definition: A node v in a time extended resource graph R_{II} is said to be *associated with* an operation i in the DFG if v is a PE executing operation i at t or v is a register in a PE that stores the result of operation i and is available at time t. For instance, in Figure 3(c) PE_2^{i+1} (PE_2 at times $i+1$) is associated with operation f^1 (operation f of first iteration), PE_2^{i+2} is associated with operation b^2, and PE_2^{i+3} is associated with operation f^1. The first register of PE_2^{i+1} is not associated with any operation, but the first register of PE_2^{i+2} is associated with operation b^2, and the first register of PE_2^{i+3} is associated with b^2.

Now, given a DFG $I = (V_I, E_I)$ and a CGRA, the problem is to construct a time extended resource graph $R_{II} = (V_R, E_R)$ of minimum extension for which:

1. there exists a surjective mapping $M : V_R^* \rightarrow V_I$, where $V_R^* \subseteq V_R$, and

2. for every arc in $(i, j) \in E_I$, the following property holds: *for each node $r_n \in R_{II}$ associated with j, there is a path $P = (r_1, \ldots, r_\ell, \ldots r_n)$ such that r_1 is a PE associated with i, r_2 through $r_{\ell-1}$ are associated with i, the rest are associated with j, and r_ℓ is a PE.*

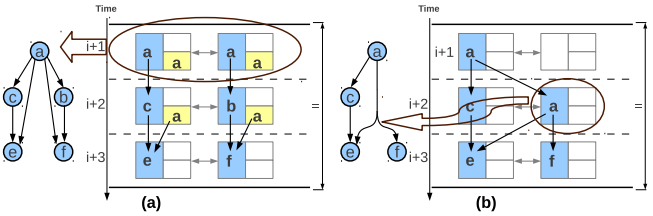


Figure 4: Problem Formulation: Our problem definition allows both routing and recomputation by allowing multiple PEs to map to the same operation. (a) If the PEs associated with same operations connected through a path, then it is routing, and if there is no path between them, then it is recomputing. operation a in this mapping is recomputed because it is associated with PE_1 and PE_2 at time i+1 and there is no path between them. (b) Our problem definition allows for path sharing.

The succinct statement of the problem makes it difficult to understand. First note that the mapping is counter-intuitive: it is from the time-extended resource graph to the DFG and not vice-versa, which is typically easier to understand. Second, the mapping is from subsets of the time-extended resource graph to an operation in the DFG. In the simplest case, the subset can be a single PE. Consider mapping presented in Figure 3. For example, PE_1^{i+1} is mapped to a^2. In a more general case, several resources may be mapped to an operation. For example, PE_2^{i+2}, the first register of PE_2^{i+2} and PE_2^{i+3} are mapped to operation b^2. The mapping for operation f^1 is also from a subset of resources (PE_2^{i+1}, and PE_2^{i+3}, and the second register of PE_2^{i+1} and PE_2^{i+2}). Third thing to note is that the mapping is surjective. This simply implies that every operation in the DFG is included in the mapping.

The second condition in the definition essentially says that if there is a dependency between two operations, then there must be a path between the subsets to which the two operations are mapped. Thus, if we pick any resource r_n which is associated with the destination operation, then a path must exist from r_1 to r_n through some r_ℓ, such that r_1 is a PE associated with the source operation (i.e., source operation is executed here), and r_ℓ is a PE associated with the destination operation (i.e., destination operation is executed here). The path between r_1 to r_ℓ can be formed of arbitrary combination of PEs and registers all associated with source operation (routing). Also, the path after r_ℓ up to r_n can also be composed of arbitrary combination of PEs and registers all associated with destination (routing).

An example of the simple case is the arc between a^2 and c^2 in Figure 3(c). For this arc, only PE_1^1 is mapped to a^2, and PE_1^2 is mapped to c^2. The path PE_1^1, PE_1^2 meets the criterion, where $r_1 = PE_1^1$, and $r_\ell = PE_1^2$. A more complicated case is the arc between a^2 and b^2. PE_1^1 is mapped to a^2, but the subset consisting of PE_2^{i+2}, and the first register of PE_2^{i+2} and PE_2^{i+3}, is mapped to b^2. The definition requires that from any resource which is mapped to b^2, there must exist a path that satisfies the second criterion. Let's take r_n to be the first register of PE_2^{i+3}. The the path (PE_1^{i+1}, PE_2^{i+2}, first register of PE_2^{i+2}, and first register of PE_2^{i+3}) satisfies the criterion, with $r_1 = PE_1^{i+1}$, and $r_\ell = PE_2^{i+2}$.

This problem definition is quite general, and allows for routing, recomputation [11], and path sharing between multiple dependencies of the same operation [5]. Routing is allowed by second condition of the definition, according to which, for each dependency, a path of resources must exist, connecting the source and destination of the dependency. Most problem definitions do not allow recomputation, since they require an operation to be mapped to one PE. As shown in Figure 4 (a), our problem definition allows for recomputation by allowing multiple PEs to be mapped to one operation. Hamzeh et al. [11] show that recomputation can result in better mappings that result in up to 2X better performance. Finally, our problem definition also allows for sharing the paths by multiple

dependencies of an operation. As Figure 4 (b) shows, if an operation has two dependents, then the resources that the dependencies use can be shared between the paths. Chen et al. [5] demonstrate that path sharing can improve resource utilization by more than 50%. Although path sharing can also be done in the methods of DRESC[6] and EMS[22], it is not explicit aspect of the solution method.

4. OVERVIEW OF OUR APPROACH

The general CGRA mapping problem is NP-complete [11]. Even so, another major challenge is that size of time-extended resource graph of the CGRA can grow very large, even for small problem sizes. For a $m \times m$ CGRA, with r registers in each PE, if we intend to map a DFG with an $II = i$, the time-extended resource graph will have $N = m \times m \times r \times i$ nodes[2]. The problem then is to find a $n+1$-partition[3] of this graph, where n is the number of operations in the DFG. The number of possible solutions to this problem is a Sterling number [10] ($\{\frac{N}{n}\}$), which is of the order of $O(n^N)$ even when the input DFG is fixed (no extra operations to be added to DFG). Due to the explosive growth of the number of partitions of time-extended resource graph, greedy approaches that attempt to find a mapping by incrementally exploring the search space are computationally very slow and result in poor solutions, i.e. high values of II. The method developed here is a constructive solution that avoids the explosive growth of the search space.

We partition the mapping problem into two sub-problems: **1) scheduling**, and **2) integrated placement and register allocation** (referred to as placement). The routing is implicitly accomplished as part of scheduling and placement. This separation of the problem into scheduling and placement results in a significant reduction of the search space. Since the operations of the scheduled DFG are bound to be placed at matching time slot in the time-extended resource graph, the number of ways to partition is $\{\frac{m}{n'}\}^i$, where n' is the maximum number of operations scheduled at any time (width of the graph). A more important advantage of breaking this problem into scheduling and placement is that *instead of explicitly enumerating the registers as nodes, we put the number of registers required as arcs weights*. This further reduces the search space by a factor of r, and makes the problem more scalable. Although the partitioning into two subproblems is in general, not optimal, we can further improve the results by performing placement and scheduling in a loop. *If a placement is not possible, then we learn from the failure, identify which operations could not be placed, and then constructively change the schedule of DFG and place them at higher priority in the next round.*

5. REGIMAP

Let $D = (V_D, E_D)$ be the scheduled graph. When operations are scheduled and II is extracted, the $R_{II} = (V_R, E_R)$ can be constructed. R_{II} is time-extended PE graph (without registers).

Step 1: Construct a compatibility graph between D and R_{II}: The compatibility graph $P = (V_P, E_P)$ is a subgraph of the product of the DFG and the R_{II}. It is a directed graph. Its vertices, which are a resource-operation pair, represent the possible mappings of operations to PEs. Its edges (ignoring the directions) indicate *compatibility* between the two corresponding mappings. Scheduling reduces the size of compatibility graph because some resource-operation pairs become incompatible, i.e. the resource is not available at the time the operation is scheduled. For instance, the number of vertices in the product graph is 16, whereas the number of vertices in the compatibility graph shown in Figure 5 is only 8. An edge between two mapping means that both can co-exist in a solution. Details of the construction of the compatibility graph is given in Appendix A.

[2] $m \times m$ PEs each with r registers is replicated i times.

[3] the $(n+1)^{th}$ partition maps to no operation

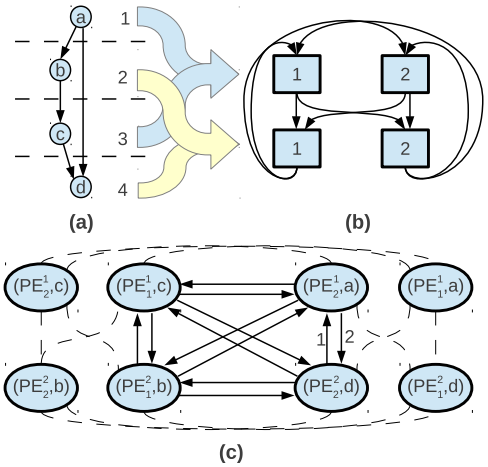


Figure 5: (a) scheduled DFG, D (b) time extended PE graph, R_2 (c) Compatibility graph P between D and R_2.

Step 2: Assign weights to arcs of compatibility graph: The weight of an arc (directed edge) denotes the number of registers required from the time the source node (a mapping) is executed to the time the destination node is executed. Note this is asymmetric. For instance in Figure 5, the arc $[(PE_2^1, a), (PE_2^1, d)]$ has weight 2, which means that from the time a is executed on PE_2 to the time that d is executed on PE_2, two registers are required in PE_2 because the $II = 2$(see Figure 2). The arc $[(PE_2^1, d), (PE_2^1, a)]$ has a weight of 1, because when d is executed, one register can be released (see Figure 2). The process of arc weight assignment in the compatibility graph is described in detail in Appendix B.

Step 3: Find maximal clique in the compatibility graph: The weight of a node is the sum of the weights of its outgoing arcs. We reduce the problem of finding a placement of D onto R_{II} to the problem of finding the largest clique whose total node weight is less than the number of available registers. By construction, the maximal clique in the graph can be no larger than $|V_D|$. Therefore, if we find a maximal clique of size $|V_D|$, in which the sum of the node weights is less than the number of registers in each PE, we have a mapping. The algorithm to find maximal clique in the compatibility graph is described in Appendix section D.

A mapping can fail because either a node in the clique does not satisfy the register constraint, or the maximal clique is smaller than $|V_D|$. In the first case, we search for another clique. If no such clique exists, or we find that an operation is not present in the clique, we reschedule that node and proceed to find a new mapping. After we run out of rescheduling options, we reduce the width of the input DFG (max. no. of operations at any cycle), and try again. If the MII increases as a result of thinning the graph, we increase II. Details of our algorithm are described in Appendix E.

6. EXPERIMENTAL RESULTS

We have modified backend GCC and integrated REGIMap right before register allocation. Loops are selected among digital signal processing applications and spec2006 benchmark suite. We compare REGIMap with register-aware DRESC [6]. It has been shown in [22] that DRESC [19] without register allocation [6] maps loops at lower II than [22], and [23].

We conduct experiments on CGRAs with different number of PEs and registers. We assume all registers are rotating registers. In our experiments, CGRA has enough memory to hold the instructions as well as variable in the loops. Since CGRA executes loops,

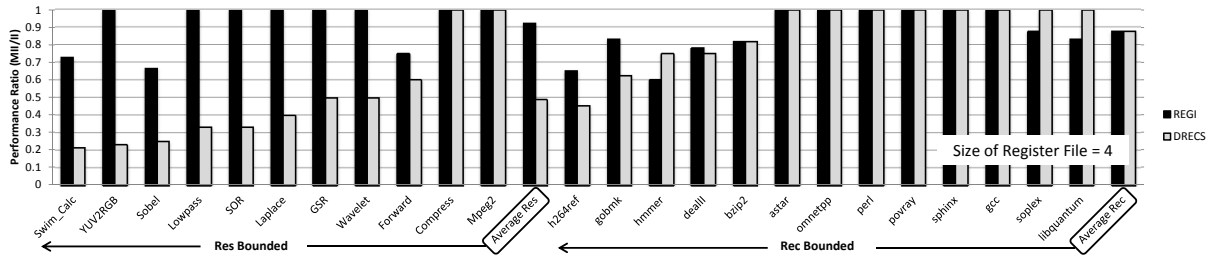


Figure 6: The performance of mapping loops to a 4×4 CGRA with 4 registers using REGIMap vs. DRESC. REGIMap compiles *Res-bounded* loops at on average 1.89 times lower II than DRESC.

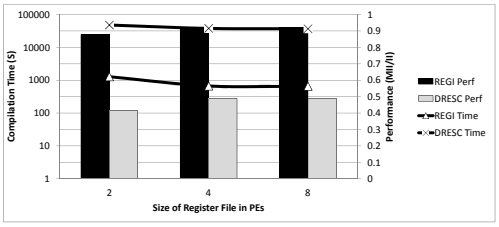


Figure 7: Compilation time using REGIMap vs. DRESC for mapping loops on 4×4 CGRA when the size of register file varies from 2 to 8.

and instructions within the kernel repeat every II cycles, a highly optimized multi-way cache can provide such memory requirement. PEs are connected in a 2D mesh-like network similar to Figure 1. We also assume that all PEs have access to the data memory but data bus is shared among PEs in a row; thus, only one PE of a row has access to the data memory at each cycle. For load and store operations, two instructions should be executed, the first instruction generates the address and the second one generates/loads data. For memory operations data to be loaded/stored is read/generated at the same row where the address bus has been asserted. The CGRA is homogeneous and PEs are capable of performing logical and fixed-point operations. All operations have latency of 1 cycle.

6.1 1.89 Times Better Performance for Resource bounded Loops

In modulo scheduling, MII is theoretically the lower bound of II for scheduled loops. Our performance metric to evaluate mappings is MII divided by the achieved II at each loop. We have divided the loops into two groups, in the first group, MII is primarily limited by ResMII, referred to as *res-bounded*. In this type of loops, placement is more challenging because the number of resources at this type of loops is relatively close to the number of operations at loop. In the second group, MII is limited by recurrence cycles (RecMII) in the DFG, referred to as *rec-bounded*. Therefore, resources are under-utilized and there are more options for each operation to be placed at. In Figure 6, the performance comparison of different applications compiled for a 4×4 CGRA with 4 registers using REGIMap and DRESC is presented.

Our evaluation reveals that on average the performance of compiled loops using REGIMap is 1.89 times higher that using DRESC technique for *res-bounded* loops. Placement in these applications is more complicated and time consuming than *rec-bounded* loops. However, the average performance of compiled loops using both mapping techniques is relatively close for *rec-bounded* loop. More details about the performance of compiled loops at different CGRA and register file sizes is given in Appendix G.

6.2 56 Times Lower Compilation Time for Resource bounded Loops

The compilation time of loops on REGIMap is much lower than compilation time of them on DRESC. DRESC is a simulated annealing (SA) based technique which spends a lot of time to move operations at time and resource dimensions. For *res-bounded* loops, REGIMap compiles them on average 37 times faster than DRESC when the size of register file of PEs is 2. The ratio of compilation time of REGIMap to DRESC increases to 56 times faster when the size of register file of PEs increases to 4 and 8. For *rec-bounded* loops, the compilation time using REGIMap is about 6 times faster than DRESC when the size of register file of PEs is 2. REGIMap becomes 8 times faster than DRESC when the size of register file increases to 4 or 8. For *rec-bounded* loops, placement is relatively easy because there are many free (not utilized) resources available in the resource graph. Those resources can be utilized for routing data between operations making placement easier. Thus, DRESC faces with less number of failed attempts which significantly decreases its running time. Both techniques compile loops faster when the size of register file increases. It is primarily because both methods face with less failure at register allocation. More details about compilation time is presented in Appendix G.

6.3 Learning from Failure Improves Performance and Reduces Compilation Time

More detailed evaluation reveals the effectiveness of operation rescheduling after placement in REGIMap. When a mapping attempt at a given II fails, most existing mapping techniques increase II. It is basically because if they schedule the DFG again and initiates a new placement attempt, they will end up with another failure. The reason is that almost all scheduling techniques use a justifiable *static* policy for ordering the nodes. Operations are to be selected by that order for scheduling. Because this ordering is static, for a DFG, the scheduling of the graph does not change unless the structure of the graph changes or the nodes are ordered differently.

The first heuristic is to change the order of the nodes. REGIMap changes this order and consequently schedule the nodes so that the next schedule is different from the previous one. Therefore, it exploits both time and resource dimensions in a guided manner.

The second heuristic REGIMap applies is virtually reducing the number of available resources at CGRA. Therefore, after many placement failure, when this heuristic is applied, the schedule of DFG further changes. Therefore, REGIMap constructively changes the schedule or the DFG to find a mapping.

Our experiments show that the rescheduling heuristic is more effective on *res-bounded* loops. To demonstrate rescheduling effectiveness, we have disabled rescheduling from REGIMap. About 90% of loops are compiled at higher IIs than when compiled with rescheduling enabled. For *rec-bounded* loops, this heuristic becomes less effective and about 30% of loops are compiled with higher IIs when it is disabled. Our further investigation reveals that most of those 30% of loops has close ResMII and RecMII.

6.4 Scalability

Figure 7 shows the average performance of compiling resource-bounded loops for a 4×4 CGRA using REGIMap and DRESC versus different sizes of register file of PEs. The left Y axis is the

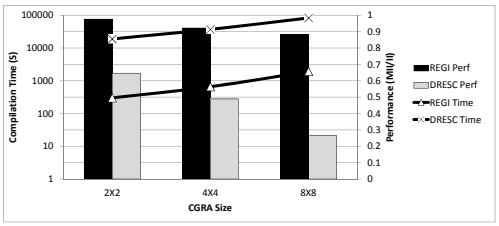


Figure 8: Compilation time using REGIMap vs. DRESC for mapping loops on 4×4 CGRA when the size of register file varies from 2 to 8.

compilation time (logarithmic) and the left one is the performance. The compilation time of loops using REGIMap and DRESC relatively decreases by increasing the size of register files. The performance of those mapping also increases when more registers are available. It is primarily because when the number of register increases, both technique can utilize more registers to compile loops. However, the performance of compiling loops using REGIMap is significantly higher (1.5X to 1.9X) compared to DRESC.

In Figure 8, the performance of compiling resource-bounded loops using two mapping techniques for different CGRA sizes when the size of register file of PEs is two is presented. The left Y axis is the compilation time (logarithmic) and the left one is the performance. The performance of loops using both techniques decreases when the size of CGRA increases. However, the performance of those mapping using DRESC dramatically decreases while the compilation time increases. The compilation time of those loops using REGIMap increases as well because the search space is proportional to the CGRA size too.

6.5 Power-Efficiency Estimation

In this section, we discuss power efficiency improvement when compute intensive loops are offloaded to the CGRA. The synthesis information presented in [3] shows that ADRES CGRA at 312 MHz consumes only 81 mW. REGIMap compiles *res-bounded* loops and achieves average of 10.75 instruction per cycle (IPC). Further details about IPC of compiled loops is presented in Appendix G. Therefore, it can be estimated that CGRA would approximately execute 3.3 GOps per second. This implies only 24 pW per instruction for CGRA and compared to 12 nW for an Intel Core2 processor [16]. Therefore, by offloading highly parallel regions in the code, instructions can be executed at approximately 500 X less energy. Considering maximum of 2 instruction per cycle for Intel Core2 processor and clock frequency of 2.6 GHz, 5.2 G instructions can be executed per second. Assuming each instruction is equivalent to one operation, we conclude that the power efficiency can approximately improve by 250 X when those loops are offloaded.

7. SUMMARY

Coarse-Grained Reconfigurable Architectures (CGRAs) are extremely attractive platform when both performance and power efficiency are paramount. However, the achievable performance and power efficiency of CGRAs critically hinges upon compiler capabilities. One of the main challenges in CGRA compilers is to efficiently utilize registers which is specially difficult due to their distributed nature. This paper make three contributions: i) we formulate the problem of mapping loops on CGRAs while efficiently using registers, ii) we present a unified and precise formulation of the problem of simultaneous placement and register allocation, and iii) an efficient and effective heuristic solution, REGIMap is distilled from our problem formulation. REGIMap leads to an average of 1.89 X better performance on several kernels from multimedia and SPEC2006 benchmarks suits when compared to the existing state-of-the-art compilation technique at lower compilation time.

8. ACKNOWLEDGMENTS

This work was partially supported by funding from National Science Foundation grants CSR-EHS 0509540, CCF-0916652, CCF 1055094 (CAREER), NSF I/UCRC for Embedded Systems (IIP-0856090), Center for Embedded Systems grant DWS-0086; Science Foundation Arizona (SFAz) grant SRG 0211-07, Raytheon, and by the Stardust Foundation.

9. REFERENCES

[1] BANSAL, N., GUPTA, S., DUTT, N., NICOLAU, A., AND GUPTA, R. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *Proc. DATE* (2004), pp. 474–479.

[2] BECKER, J., AND VORBACH, M. Architecture, memory and interface technology integration of an industrial/ academic configurable system-on-chip (csoc). In *Proc. ISVLSI* (2003), pp. 107–112.

[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. C. *Programmable arithmetic devices for high speed digital signal processing*. PhD thesis, University of California, Berkeley, 1992.

[5] CHEN, L., AND MITRA, T. Graph minor approach for application mapping on cgras. In *Proc. FPT* (2012).

[6] DE SUTTER, B., COENE, P., VANDER AA, T., AND MEI, B. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *Proc. LCTES* (2008), pp. 151–160.

[7] DIMITROULAKOS, G., GALANIS, M., AND GOUTIS, C. Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures. In *Proc. IPDPS* (2006), pp. 113–122.

[8] FRIEDMAN, S., CARROLL, A., VAN ESSEN, B., YLVISAKER, B., EBELING, C., AND HAUCK, S. Spr: an architecture-adaptive cgra mapping tool. In *Proc. FPGA* (2009), pp. 191–200.

[9] GOLDSTEIN, S., SCHMIT, H., MOE, M., BUDIU, M., CADAMBI, S., TAYLOR, R., AND LAUFER, R. Piperench: a coprocessor for streaming multimedia acceleration. In *Proc. ISCA* (1999), pp. 28 –39.

[10] GRAHAM, R. L., KNUTH, D. E., AND PATASHNIK, O. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[11] HAMZEH, M., SHRIVASTAVA, A., AND VRUDHULA, S. Epimap: using epimorphism to map applications on cgras. In *Proc. DAC* (2012), pp. 1284–1291.

[12] HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. In *Proc. DATE* (2001), pp. 642–649.

[13] HARTENSTEIN, R., AND KRESS, R. A datapath synthesis system for the reconfigurable datapath architecture. In *Proc. ASP-DAC* (1995), pp. 479 –484.

[14] HATANAKA, A., AND BAGHERZADEH, N. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Proc. IPDPS* (2007), pp. 1–8.

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

[16] KEJARIWAL, A., VEIDENBAUM, A., NICOLAU, A., TIAN, X., GIRKAR, M., SAITO, H., AND BANERJEE, U. Comparative architectural characterization of spec cpu2000 and cpu2006 benchmarks on the intel core 2 duo processor. In *Proc. SAMOS* (july 2008), pp. 132 –141.

[17] LEE, J.-E., CHOI, K., AND DUTT, N. D. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design and Test of Computers 20*, 1 (2003), 26–33.

[18] 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 computingprocessor. *J. VLSI Signal Process. Syst. 24* (2000), 147–164.

[19] MEI, B., VERNALDE, S., VERKEST, D., DE MAN, H., AND LAUWEREINS, R. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In *Proc. IEEE FPT* (dec. 2002), pp. 166 – 173.

[20] 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.

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

[22] 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 *Proc. PACT* (2008), pp. 166–176.

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

[24] TAYLOR, M. B. Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In *Proc. DAC* (2012), pp. 1131–1136.

[25] YOON, J., SHRIVASTAVA, A., PARK, S., AHN, M., AND PAEK, Y. A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Trans. on VLSI Systems 17*, 11 (2009), 1565–1578.

APPENDIX

A. COMPATIBILITY GRAPH CONSTRUCTION

Construction of Compatibility Graph $P = (V_P, E_P)$ is completed in two steps. In the first step, the set of nodes in this graph is formed. Then directed unweighed arcs between nodes are created.

A.1 Nodes

First we define the Cartesian product of set of nodes in graph R_{II} and D. Every element in this set represents a pair of an operation and a resource in R_{II}. Basically a pair of nodes represents potential mapping of an operation to a resource. Since operations at graph D are scheduled, we can easily reject a large number of potential mapping of operations represented by some elements in CP. For instance, consider $u_i = (v_i^R, v_i^D) \in CP$. This potential mapping of operation v_i^D on v_i^R can be rejected under the following conditions. If the ALU of v_i^R does not support operation of v_i^D, then the mapping of this of v_i^D to v_i^R is not feasible. Similarly, if v_i^R is not present at the cycle when v_i^D is scheduled, then u_i is not a potential solution. Based on observation above, we create the set of nodes in P.

Let $CP = \{V_R \times V_D\}$ (Cartesian product). The set of nodes in P is a subset of CP. Thus, each node $u_i = (v_i^R, v_i^D) \in CP$ represents a pair of a resource[4] at R_{II} and an operation. We define v_i^R and v_i^D compatible if resource v_i^R supports (its ALU) the boolean function of operation v_i^D and it is present at the same cycle (in R_{II}) that the operation is scheduled. In Figure 2(c), let $PE(t, n)$ represents PE_n at cycle t in the resource graph. Assume that operation a is scheduled at cycle 1. Then pair $(PE(1, 2), a)$ is a compatible pair because $PE(1, 2)$ is present at the cycle 1. None of pairs $(PE(2, 1), a)$ nor $(PE(2, 2), a)$ are compatible because they are not present at time 1.

A.2 Binary Arcs

Next, REGIMap constructs the set of arcs in P. In this step, we define a reflexive and symmetric relation, called compatibility, between pairs of nodes in V_P. Note that each element in V_P represents a potential mapping of a node to a resource. compatibility between elements $u_i = (v_i^R, v_i^D)$ and $u_j = (v_j^R, v_j^D)$ in V_P essentially implies that both mapping can co-exist in the solution. More precisely, when v_i^D is mapped to v_i^R, does it restrict v_j^D from being mapped to v_j^R. If there is an arc from u_i to u_j, then if u_i is in a solution mapping, u_j can also be in that solution.

Two nodes u_i and u_j where $i \neq j$ are incompatible if one of the following conditions holds:

1. u_i and u_j represent the same operation ($v_i^D = v_j^D$).
2. u_i and u_j represent the same resource ($v_i^R = v_j^R$).
3. There is an arc between the operation u_i represents to the one u_j represents, but there is no communication from resources u_i represents to u_j represents.

There are two considerations for the 3rd case. v_i^D and v_j^D can be scheduled apart (more than one cycle apart) and there is no arc in R_{II} to connect resources apart more than one cycle. For this case, REGIMap tags u_i and u_j incompatible if they represent different PEs in the CGRA.

[4]We do not use the term PE to avoid confusion between physical PE or replicated PEs in R_{II}

The second problems arises when there is an inter-iteration data dependency between operations u_i and u_j represent. Similar to the previous case, REGIMap tags them incompatible, if they represent different PEs in the CGRA.

For example, consider the following nodes $(PE(1, 2), a)$ and $(PE(2, 1), b)$ in Figure 2(d). These nodes are compatible because there is an arc from a to b in DFG as well as an arc between $PE(1, 2)$ and $PE(2, 1)$. However, $(PE(1, 2), a)$ and $(PE(1, 1), a)$ are not compatible because they both represent operation a. On the other hand, $(PE(3, 2), a)$ and $(PE(4, 2), d)$ are compatible[5] because both nodes represent the same physical PE at CGRA where a and d are scheduled 3 cycles apart.

When there is a symmetric compatibility between u_i and u_j, two directed arcs between them are to be formed. Next, REGIMap looks for a pair of nodes representing two operations with intra-iteration data dependency scheduled more than one cycle apart. Next, weigh of arcs needs to be assigned.

B. ARC WEIGHT ASSIGNMENT TO THE COMPATIBILITY GRAPH

The weight of arcs represent the number of registers required to establish a path between a pair of mappings. Let $T : V_D \rightarrow N$ be the schedule function. Consider $u_i = (v_i^R, v_i^D)$ and $u_j = (v_j^R, v_j^D)$ where there is an arc between v_i^D and v_j^D and they are not scheduled in two consecutive cycles. The weight of arc between u_i and u_j is to be increases by:

$$R = \left\lceil \frac{T(v_j^D) - T(v_i^D)}{II} \right\rceil \quad (1)$$

When these operations are scheduled less than II cycles, one register is sufficient to carry-out data dependency between them. However, when they are scheduled more than II cycles apart, this data dependency must be carried-out across multiple iterations of the loop. Thus, when a new iteration starts, when v_i^D of the next iteration of the loop is executed, if the previous output has not been consumed yet, the output of this operation must be stored in a different register. Otherwise, it overwrites a register that has not been read yet by the consumer operations.

Note that a register is allocated when v_i^D is executed on v_i^R until v_j^D is executed on v_j^R at all resources representing same PE as v_i^R represents. Therefore, the weight of arcs from all u_k representing those resources to v_i^R must be increased by R. At $T(v_j^D)$, a register is to be released at v_j^R until a new data is produced by v_i^D at the next iteration of the loop. Thus the arc from nodes representing resources between v_j^R and v_i^R to v_i^R must be increased by $R - 1$.

In Figure 2(d), operations a and d are scheduled more than one cycle apart. The weight of arc from $(PE(3, 2), a)$ to $(PE(4, 2), d)$ should to be increased by 2 [6]. However, the weight of arc from $(PE(4, 2), d)$ to $(PE(3, 2), a)$ is only increased by 1 because one register is to be released when d is executed.

Operations with inter-iteration data dependencies are to be considered next. Let $u_i = (v_i^R, v_i^D)$ and $u_j = (v_j^R, v_j^D)$ represent two operations with inter-iteration data dependency. Let $e_{(v_i^D, v_j^D)}$ be the inter-iteration distance of these operations. The weight of arc

[5]$II = 2$

[6]$\lceil \frac{4-1}{2} \rceil = 2$

between u_i and u_j is to be increases by:

$$R = e_{(v_i^D, v_j^D)} - \begin{cases} \left\lceil \frac{T(v_j^D) - T(v_i^D)}{II} \right\rceil - 1 & \text{if } T(v_j^D) \geq T(v_i^D) \\ \left\lfloor \frac{T(v_j^D) - T(v_i^D)}{II} \right\rfloor & otherwise \end{cases} \quad (2)$$

When v_j^D is scheduled after v_i^D and $T(v_j^D) - T(v_i^D) > II$, because of the repetitive execution of the loop, the inter-iteration distance between them decreases because they belong to different iteration of the loop at execution. On the contrary, when v_i^D is scheduled after v_j^D, by increasing the schedule distance, their inter-iteration distance increases.

Similar to the previous case, at all resources between v_i^R to v_j^R, R registers should be available to establish a path between v_i^R to v_j^R. Therefore, the weight of arcs from all nodes u_k representing above-mentioned resources to u_i must be increased by R. However, the weight of arcs from nodes representing the resources between v_j^R to v_i^R that represent same PE of v_i^R to u_i must be increase by $R-1$. This step also considers operations with self inter-iteration data dependency. Note that the weight of arcs between two node in P is to be update only if those nodes are compatible.

C. PROBLEM REDUCED TO NODE TOTAL WEIGHT CONSTRAINED MAXIMUM CLIQUE

THEOREM C.1. *In graph P, there is a maximum clique C (of size $|V_D|$) such that the sum of the weight of outgoing arc is less than the size of register file,i.e. $\forall u_i \in V_C : S_{u_i} < N_R$ where $S_{u_i} = \sum_{\forall (u_i,u_j) \in E_C} \{e_{(u_i),(u_j)}\}$ and where N_R is the size of register file of PEs, iff there is a placement of operation so that the maximum of N_R registers are used at each PE.*

D. FINDING MAXIMAL CLIQUE ALGORITHM

REGIMap conducts a greedy strategy to find a clique in P. Although P is a directed graph, the arcs between nodes are formed symmetrically though the weight of arcs can be different. Thus, finding a clique in P is similar to finding a clique in an undirected graph.

To find a clique, REGIMap starts with an empty clique. To maximize the clique size, REGIMap finds a node that is connected to all nodes present at clique. In the case of many, REGIMap selects the one with the maximum number of arcs to the nodes outside the clique. This heuristic is applied to increase the chance of maximizing the clique size. When the size of a clique cannot be further increased, if finds a nodes outside the clique connected to all nodes inside the clique expect one. In that case, REGIMap replaces this new node with the old one and tries to further increase the size of the clique. When this condition is checked for all nodes at P REGIMap stores this clique in a list.

REGIMap later finds the intersect of pairs of cliques. This intersect is the next initial clique to be maximized. At the end, REGIMap returns the maximum clique it finds after the above steps. At every step to increase a clique size, when a nodes is selected, the sum of the weights of outgoing arcs from the selected node to all nodes in clique is checked to be less than N_R. If a node violates this condition, it cannot be added to the clique. Note that during intersection of cliques, there is no need to verify the number of allocated registers.

Once the size of the clique increases to $|V_D|$, the placement is complete. A clique at the end of this step, represents the placement of operations and the sum of weight of outgoing arcs of a node, determines how many registers are used at all PEs. If the size of maximum clique is less than $|V_D|$, some nodes in D could not be placed, thus REGIMap, change those nodes at the next step.

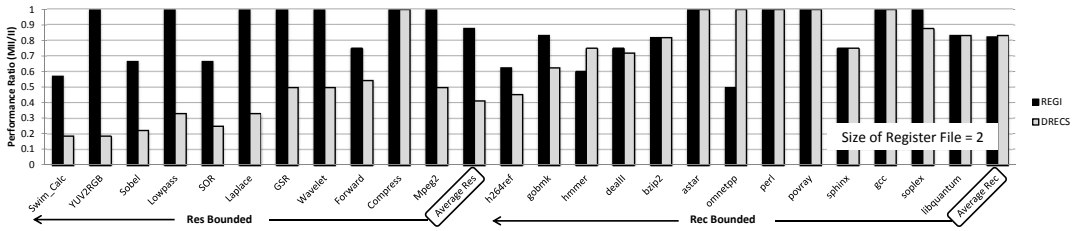
E. REGIMAP ALGORITHM

REGIMap initially extracts the minimum II using technique in [15]. Then operations are scheduled with the goal of minimizing II. In the next step, REGIMap constructs a time extended resource graph. In this graph, only PEs are present. Afterwards, a compatibility graph P is to be 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.

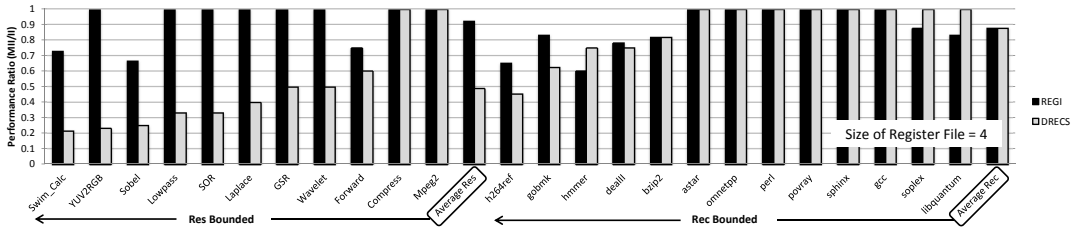
Algorithm 1: REGIMap(Input D, Input $CGRA$)

```
begin
    MII ← DetermineMII(D, |V_D|);
    S ← |V_C|; D_s ← D;
    while true do
        N ← ∞;
        while true do
            D_S, II ← Schedule(D_S, S);
            if II > MII then
                MII ← MII + 1;
                S ← |V_C|; D_S ← D;
                break;
            R_II ← Construct_Resource_Graph(C, MII);
            P ← Construct_Compatibility_Graph(D_S, R_II);
            C ← Weight_Constrained_Max_Clique(P);
            if |V_C| = |V_Ds| then
                Return C;
                else
                    if |V_DS| − |V_C| > N then
                        S ← S − 1; D_S ← D;
                        break;
                        else
                            D_S ← Re-Schedule(V_DS − V_C);
                            N ← |V_DS − V_C|;
```

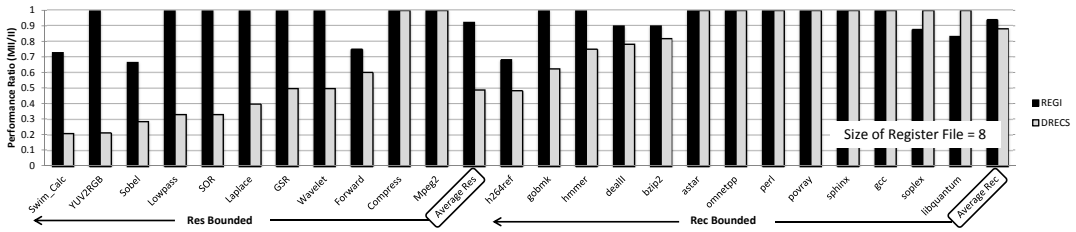
Re-Scheduling: If a placement fails to place and allocate registers for some operations, the set of unplaced operations are rescheduled. REGIMap reschedules those operations to one cycle earlier than their current schedule cycle, or insert extra routing nodes if failed operation requires register for placement (relaxing routing problem). If the schedule cycle of an operation is decreased to a cycle lower than least feasible one (determined by scheduling), or adding extra nodes increases lower bound II, then the DFG is to be rescheduled with a new heuristic. At this step, the number of available PEs is set to be $N-1$ where N was the number of available PEs at the previous scheduling attempt. N is initially set to the number of PEs. Please note that decreasing the number of available resources can increase II at scheduling. In such a case, REGIMap



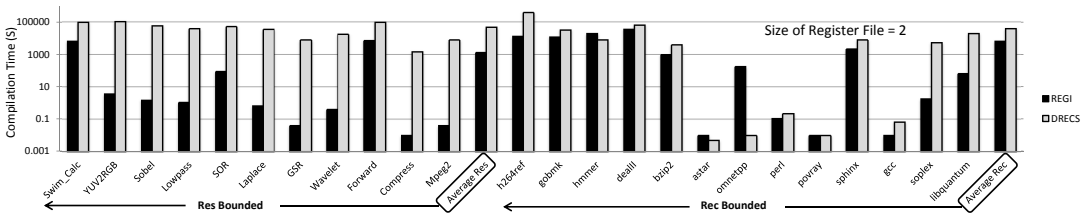
(a) Performance of compiled loops using REGIMap and DRESC for a 4×4 CGRA with 2 registers at each PE



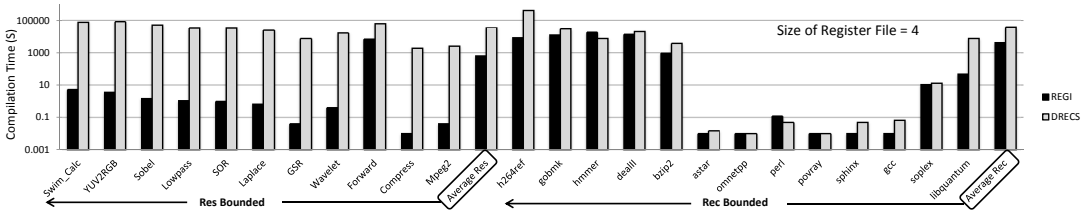
(b) Performance of compiled loops using REGIMap and DRESC for a 4×4 CGRA with 4 registers at each PE



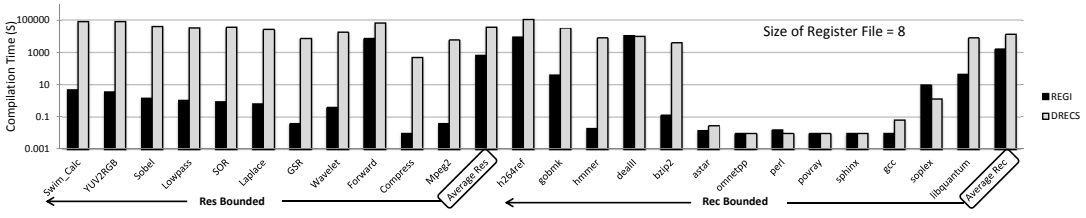
(c) Performance of compiled loops using REGIMap and DRESC for a 4×4 CGRA with 8 registers at each PE



(d) The compilation time of loops using REGIMap and DRESC for a 4×4 CGRA with 2 registers at each PE



(e) The compilation time of loops using REGIMap and DRESC for a 4×4 CGRA with 4 registers at each PE



(f) The compilation time of loops using REGIMap and DRESC for a 4×4 CGRA with 8 registers at each PE

Figure 9: The performance of compiling and compilation time of loops using REGIMap versus DRESC for different size of register file at PEs.

increases MII by one and reset the number of available resources to be the number of PEs. When MII increases, REGIMap proceeds with a new scheduling and placement attempt.

F. HOW IS THE PROBLEM OF REGISTER ALLOCATION ON CGRA DIFFERENT THAN ON VLIW ARCHITECTURES?

Register allocation in the problem of mapping application to VLIW architectures is essentially different from CGRAs. In a VLIW architectures, the register file is central providing connection to all functional units at a high bandwidth. Therefore, when dependent operations mapped to separate resources, data dependency between them can readily satisfied using central register file. In CGRA, however, the register files are distributed along with functional units or PEs. When two dependent operation are mapped to separate resources, data should explicitly routed between them. Therefore, communication cost is proportional to the distance of resources where dependent operation are mapped. This implies a tight dependency between placement and register allocation in CGRA application mapping while it is not the case for VLIW architectures where communication cost is approximately constant.

If registers are to be allocated after scheduling and placement in CGRAs, in the case of a failure in register allocation, the cost of inserting spill code is to reconstruct an entire new mapping. This cost is primarily due to the modulo scheduling where the execution is repetitive. Insertion of spill code is a typical technique utilized in compilers for VLIW processor. Therefore, a different policy for register allocation is required in the area of CGRA application mapping.

G. MORE RESULTS

Application	REGIMap IPC
Swim_Calc	10.52
YUV2RGB	12.33
Sobel	9.67
Lowpass	13.5
SOR	12.5
Laplace	11.5
GSR	12
Wavelet	10
Forward	7.25
Compress	9
Mpeg2	10
Average Res- Bounded	10.75

Table 1: Instruction per cycle (IPC) of mapping of res-bounded loops of CGRA using REGIMap.