Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures

Hyunchul Park, Kevin Fan, Manjunath Kudlur, Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{parkhc, fank, kvman, mahlke}@umich.edu

ABSTRACT

Coarse-grained reconfigurable architectures (CGRAs) present an appealing hardware platform by providing the potential for high computation throughput, scalability, low cost and energy efficiency. CGRAs consist of an array of function units and register files generally organized as a two dimensional grid. The most difficult challenge with deploying CGRAs is compiler scheduling technology that can map software implementations of compute intensive loops onto the array. Traditional schedulers are not suitable because they do not take into account the explicit routing of operand values that is necessary. In essence, the problem of binding operations to time slots and resources is extended to also include explicit routing of operands from producers to consumers. To tackle this problem, this paper introduces a software pipelining technique for mapping loop bodies onto CGRAs, referred to as *modulo graph embedding*. We leverage graph embedding from graph theory, which is used to draw graphs onto a target space. The loop body is essentially drawn onto the CGRA mesh, subject to modulo resource usage constraints. Modulo graph embedding is effective because it can take into account the communication structure of the loop body during mapping. On average, a compute utilization of 56–68% is achieved for a set of loop kernels across three 4x4 CGRA designs.

Categories and Subject Descriptors

D.3.4 [**Processors**]: [Code Generators]; C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and Embedded Systems]

General Terms

Algorithms, Experimentation, Performance

Keywords

Coarse-grained Reconfigurable Architecture, Graph Embedding, Modulo Scheduling

1. INTRODUCTION

The embedded computing systems that power today's portable devices demand high performance and energy efficiency. Traditionally, application specific hardware in the form of ASICs has been used on the compute-intensive kernels to meet these demands. However, increasing convergence of different functionalities, such as voice/data communication, high definition video, and digital photography on a single device, combined with high non-recurring costs involved in designing ASICs have pushed designers towards more flexible solutions. Coarse-grained reconfigurable architectures (CGRAs) are becoming attractive alternatives because they offer large raw computation capabilities with low cost/energy implementations [13, 18, 14]. Furthermore, CGRAs are programmable, thus software implementations of compute intensive kernels can be mapped onto them.

CGRAs consist of an array of a large number of function units (FUs) interconnected by a mesh style network. Register files are distributed throughout the CGRAs to hold temporary values and are accessible only by a subset of FUs. The FUs can execute common word-level operations, including addition, subtraction, and multiplication. In contrast to FPGAs, CGRAs have short reconfiguration times, low delay characteristics, and low power consumption as they are constructed from standard cell implementations. Thus, gate-level reconfigurability is sacrificed, but the result is a large increase in hardware efficiency.

A good compiler is essential for exploiting the abundance of computing resources available on a CGRA. However, sparse connectivity and distributed register files present difficult challenges to the scheduling phase of a compiler. The sparse connectivity puts the burden of routing operands from producers to consumers on the compiler. Traditional schedulers that just assign an FU and time to every operation in the program are unsuitable because they do not take routability into consideration. Operand values must be explicitly routed between producing and consuming FUs. Further, dedicated routing resources are not provided. Rather, an FU can serve either as a compute resource or as a routing resource at a given time. A compiler scheduler must thus manage the computation and flow of operands across the array to effectively map applications onto CGRAs.

Previous efforts at compilation tools for CGRA style architectures have focused on exploiting instruction-level parallelism (ILP) [10, 1]. However, ILP is limited in scope, and fails to efficiently make use of resources in CGRAs. Recent research [14] has focused on exploiting loop-level parallelism on CGRAs. They propose a modulo scheduling algorithm based on simulated annealing. It begins with a random placement of operations on the FUs of a CGRA, which may not be a valid modulo schedule. Operations are randomly moved between FUs until a valid schedule is achieved.

The random movement of operations in the simulated annealing technique results in a long convergence time for loops with large numbers of operations. Also, the algorithm is ad-hoc in the sense that no information about the structure of the dataflow graph is utilized in making scheduling decisions.

In this paper, we propose a modulo scheduling technique for CGRA architectures that leverages graph embedding commonly used in graph layout and visualization [12], referred to as *modulo graph embedding*. Graph embedding is a technique in graph theory in which a guest graph is mapped onto a host graph. With CGRAs, scheduling is reduced to placing operations of a loop body on a three dimensional grid. The three dimensions consist of the FU array that comprises two dimensions and the time slots of a modulo scheduled loop that form the third dimension.

Modulo scheduling is performed by considering groups of equal height operations from the top of the dataflow graph (DFG) to the bottom. The three dimensional scheduling grid is filled in a skewed manner by restricting the subset of FUs and time slots available for each group of operations. This stylization increases routability of operands and can dynamically adapt to different shape DFGs. A discrete cost function between pairs of DFG nodes is designed and the placement algorithm tries to reduce this cost function. The cost function consists of different components: routing cost, which ensures that producers and consumers are placed close to one another; affinity cost, which ensures that operations with common consumers are placed close together; and, position cost, which ensures that operations are left-justified on the set of eligible resources. Left justification ensures operations are tightly packed and enables operand routing to subsequent operations using the righthand portion of the array.

The central advantages of modulo graph embedding are summarized as follows:

- It scales well with respect to number of operations in the DFG and thus is capable of handling large loop bodies.
- It handles a wide variety of CGRA configurations, including sparse interconnectivity and fully distributed register files.
- It is a systematic technique that assigns operations to the nodes in a CGRA and thus convergence to a solution is faster along with producing higher quality schedules.

2. BACKGROUND AND MOTIVATION

2.1 Architecture Overview

A CGRA consists of an array of compute nodes, each of which executes word-level operations, communicating through an interconnection network. In general, CGRA designs can be described by four characteristics: size, node functionality, network configuration, and register file sharing. The *size* refers to the number of nodes; commonly this can vary from 4 nodes arranged in a row up to 64 nodes arranged in an 8x8 grid. The *functionality* of each node can vary from a single FU (e.g. adder or subtracter), to an ALU, to a full-blown processor. In addition, the functionality of nodes may be homogeneous or heterogeneous. For example, only the nodes on the edges of the array may access data memory.

There is a large number of potential *network configurations*, such as connections between each node and its four (or eight diagonal) nearest neighbors, buses connecting each node to (possibly to a subset of) other nodes in the same row or column, hierarchical connection schemes, and so on. Finally, the degree of *register file sharing* ranges from small, individual register files at each node, to multiple register files each shared by a small number of nodes, to a single central register file accessible by some or all nodes.

Figure 1 shows three CGRA designs. Each design contains 16 nodes arranged in a 4x4 mesh; each node can communicate with its four nearest neighbors. The details of a node are shown in Figure 1(a). A FU reads inputs from neighboring nodes and writes to a single output register; a small, dedicated register file can supply operands to the FU and store the FU's result; and a configuration memory supplies control signals to the MUXes, FU, and register file. Note that a node can either perform a computation or route data each cycle, but not both, as routing is accomplished by passing data through the FU.

The CGRA designs shown in Figure 1 vary in terms of their register file sharing. In design (a), there is no register file sharing as all register storage is distributed across the nodes in the form of the dedicated register files. The interconnect in this design is therefore the most sparse of the three designs, as all communication between nodes must be explicitly routed through the network. Design (b) is an example of a shared register file design (four neighboring FUs share a register file in this example), where some nodes have access to the same register files. FUs that share a register file can communicate values directly, without explicit routing through the interconnection network. The downside of having shared register files is that the files are larger and more highly ported, reducing the efficiency of the hardware.

Finally in design (c), all nodes again have individual register files, but now they are additionally connected to a central register file via column buses. This machine is essentially equivalent to a VLIW processor where all FUs can communicate with each other by writing to the same register file. This design is richest in terms of communication abilities, but it is also the least scalable as the central register file must be large and highly ported to supply operands to all FUs. Generally, the ports on the central register file are limited to a small number to control cost and allow a subset of the inter-FU communication to go through the central register file. These three examples show only a small subset of possible CGRA designs as many other variations are possible.

For this research, we assume a 4x4 CGRA with homogeneous FUs in a mesh interconnection network as shown in Figure 1a. Even though a richer interconnect (e.g., mesh plus [14]) can make scheduling easier and usually leads to better schedules, our objective is to develop a scheduler that can achieve good performance under the restrictions of a low-cost CGRA. For the basic scheduler formulation, register files are fully distributed over a 4x4 array of FUs (e.g., no shared register files). In the experiments, we vary the register file configuration to evaluate the effectiveness of scheduling across different CGRA configurations.

2.2 Modulo Scheduling for CGRAs

Modulo scheduling is a software pipelining technique that exposes parallelism by overlapping successive iterations of a loop [16]. The goal is to find a valid schedule for a loop such that the interval between successive iterations (initiation interval, or II) is minimized. The II-cycle code region that achieves this maximal overlap is called the kernel. When the number of iterations is large, the performance of the loop is determined by the II to a first order; thus, when modulo scheduling, it is more important to minimize the II than to minimize schedule length. Initially, the scheduler chooses the target II to be the maximum of the resource-constrained lower bound (ResMII) and the recurrence-constrained lower bound (RecMII). If a valid modulo schedule cannot be found, the target II is incremented and scheduling is attempted again.

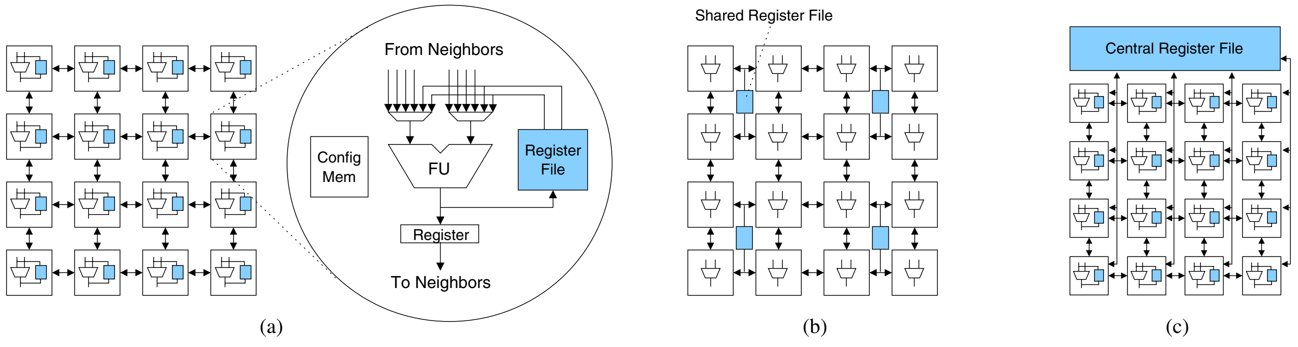


Figure 1: Three CGRA designs with varying register file connectivity: (a) dedicated register files, (b) shared register files, and (c) dedicated register files and a single centralized register file.

Scheduling for CGRAs is quite different from scheduling for general VLIW architectures due to the different hardware characteristics. With the presence of the central register file in general VLIW architectures, scheduling consists of finding available resources for producers and consumers. Routing from a producer to a consumer is implicitly guaranteed by storing intermediate values in the central register file.

However, just finding resources for computation is not sufficient when scheduling for a CGRA because of the sparse interconnect and distributed register files. The intermediate values stored in the distributed storage elements (either local register files or output registers of FUs) need to be routed explicitly to the consuming FUs. As FUs are used for both computation and routing, the routability of values has to be checked at schedule time. If this condition is not checked, a situation could arise where a value is produced by an FU but cannot be routed anywhere as all of the neighboring FUs are busy in subsequent cycles. Thus, the scheduler should be aware of these routing requirements not only to generate a valid schedule, but also to minimize the number of routing resources used so that more FUs are available for computation.

One approach to modulo scheduling for CGRA arrays was described by Mei et al. [14]. In this approach, the architecture is represented by a modulo routing resource graph (MRRG), and the goal of scheduling is to map the DFG and any necessary data transfer operations to this MRRG through time. To accomplish this, all operations are first scheduled on a subset of the FUs, over-committing those resources. Then, using a simulated annealing approach, operations are iteratively removed from the existing schedule and randomly placed on new FUs. A cost function, based on the degree resources are over-committed, is used to decide whether or not to accept each new schedule. The objective is to move towards schedules with fewer over-committed resources, and a valid schedule is found when no resources are over-committed.

The limitation of this approach is that it is not scalable with respect to the size of the DFG. With larger graphs, the schedule time increases significantly due to the time complexity of simulated annealing, and the scheduler may not converge to a solution at the desired II. In addition, the probability of convergence is lower for CGRAs with sparser network connectivity schemes, as it becomes more difficult to find a valid, routable schedule using random placement. Our approach overcomes these problems by using information available in the DFG, such as its overall shape and the relationships between producers and consumers, to systematically schedule the operations onto the CGRA.

3. MODULO GRAPH EMBEDDING

This section describes modulo graph embedding, our approach to modulo scheduling for CGRAs. We break the description down into two parts: Section 3.1 presents the important concepts of the approach in isolation, and Section 3.2 brings everything together to discuss the complete scheduling algorithm.

3.1 General Concepts

3.1.1 Resource and Connectivity Management

During instruction scheduling, a reservation table is maintained to keep track of which resources are used in each time slot. As resources are repeatedly used every II cycles by successive iterations of the loop, the modulo scheduler maintains a Modulo Reservation Table (MRT) which has only II time slots [16]. Considering that the scheduler for a CGRA must perform routing of values as well as placement of operations, routing information should be recorded by the scheduler. This routing information can be included in the reservation table because FUs are used both for computation and routing. Management of the interconnect network is not necessary as all of the connections are dedicated point-to-point connections, meaning that no congestion can occur in the network.

For resource management, the concept of the Modulo Routing Resource Graph (MRRG) from the DRESC compiler framework [14] is used. The MRRG is a graphical representation of the scheduling space where nodes represent routing resources and edges describe the connectivity of those resources. Scheduling in the CGRA becomes a problem of placement and routing of each operation on the MRRG.

The original MRRG has a detailed description of the CGRA [14]. MRRG nodes are created for each port on the FUs and register files, in addition to the MRRG nodes for the FUs and register files themselves. We take a simplified approach to model the CGRA. A single node is created for each FU and register file. Since port information for FUs can be easily discovered by analyzing the resulting schedule along with the instruction format, it is not necessary to create nodes for the individual FU ports. Port information for register files can also be discovered in the same way, but two additional nodes are used to limit the number of read/write accesses to register files. Our resource management model can be considered as a distributed MRT with connectivity information. Each node represents either an FU or register file and is equipped with a MRT to keep track of the resource usage.

Figure 2(b) shows our resource management model constructed for the 2x2 CGRA in Figure 2(a) with II = 3. Nodes for register files and wrap-around edges were omitted for simplicity. Each of

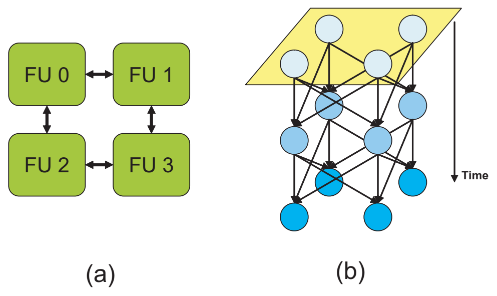


Figure 2: Modelling resources in a CGRA: (a) example 2x2 CGRA, (b) resource management model for 2x2 CGRA with II=3.

the four nodes in the CGRA has a 3-entry MRT, and each edge specifies that a value can be routed from the source to the destination resource. When an operation is placed on an FU, the MRT in the corresponding node is marked as occupied at the schedule time. If there are any placed producers or consumers, a valid route is discovered by traversing nodes along the edges.

3.1.2 *Register Assignment and Allocation*

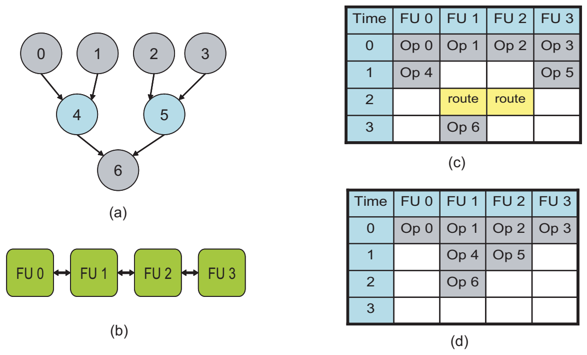
With modulo scheduling, the number of registers required by a loop is not known before scheduling. In addition to conventional register allocation constraints, it may be necessary to keep multiple copies of registers depending on how many iterations separate the first producer and last consumer. This can cause a problem for the small, distributed register files in CGRAs as the number of total registers required at a single FU can exceed the local register file capacity. The available storage must be carefully considered during scheduling as simply pushing register allocation to after scheduling can result in costly spilling and may require complete rescheduling of the loop.

Our approach is to perform a simple register allocation and assignment during modulo scheduling. The modulo constraint that is enforced for FUs is also enforced for registers, i.e., there is an MRT kept for the each register file. A register value can stay in the same register up to II cycles, but the value will be overwritten by the same instruction in the next iteration II cycles later. When a register value is live for longer than II cycles, it has to be explicitly routed to another register file (or to another register in the same file). Specific entries in the register file are allocated for each virtual register using a simple greedy algorithm. While this approach may seem overly simplistic, it effectively guides the scheduler to distribute register usage across the CGRA.

3.1.3 *Height-based Scheduling*

The problem of modulo scheduling for a CGRA can be viewed as mapping applications onto the 3-D space consisting of the FU array stacked up II times. With this finite scheduling space, minimizing the *routing cost* is a critical issue in scheduling, as fewer resources being used for routing leads to more resources being available for computation. Routing cost is defined as the number of FUs being used for routing (passing data from one node to another) rather than computation. This cost depends on the positions of producer and consumer operations in the CGRA due to the sparse interconnect network. This requires the scheduler to be cognizant of producer and consumer relations so that they can be placed close to each other.

Figure 3 shows how the placement of operations impacts the routing cost of their consumers. Figure 3(a) is an example DFG



Time	FU 0	FU 1	FU 2	FU 3
0	Op 0	Op 1	Op 2	Op 3
1	Op 4			Op 5
2		route	route	
3		Op 6		

Time	FU 0	FU 1	FU 2	FU 3
0	Op 0	Op 1	Op 2	Op 3
1		Op 4	Op 5	
2		Op 6		
3				

Figure 3: Example showing the placement of producers affects the routing cost of consumers: (a) DFG for loop, (b) target architecture which is a 1x4 CGRA, (c) poor schedule that results in an extra cycle for routing values to Op 6, and (d) good schedule that results in no additional routing.

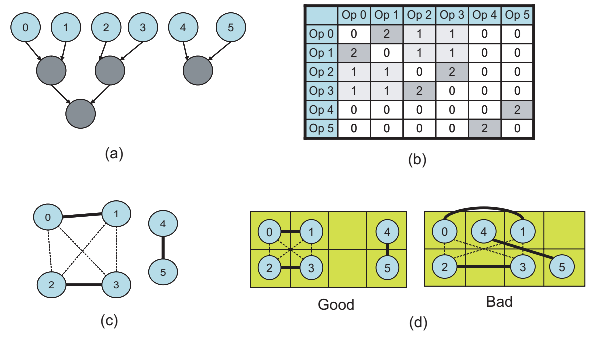
and Figure 3(b) is a hypothetical architecture with sparse interconnect where FUs are allowed to communicate only with adjacent FUs. Figures 3(c) and Figure 3(d) show two different schedules, both minimizing the routing cost for operations 4 and 5. When operation 6 is placed, the minimal routing cost is affected by the positions of its two producers (operations 4 and 5). This suggests that the scheduler must proactively choose placements to reduce routing costs.

To effectively manage routing costs, we employ two complementary techniques: height-based scheduling and the affinity-based placement which is discussed in the next section. Height-based scheduling is a common heuristic used in list scheduling where operations are scheduled in the order of dependence height. Operations with greater height are scheduled first, followed by operations with lower height. But, for operations with the same height, a CGRA scheduler cannot process them individually because placement of one operation has cost implications on the placement of others. Careless placement of one operation might increase the routing cost of other operations, or even make it impossible to place by blocking all of its routing possibilities. Therefore, operations with the same height are considered together to achieve an optimal placement rather than being scheduled separately. Possible schedule slots (resource/time pairs) are identified for each operation, and a combination of schedule slots (called a *layout*) that minimizes the total routing cost is selected.

3.1.4 *Affinity Graph*

Routing cost is difficult to minimize during scheduling because the true cost is not known until all producer-consumer pairs are placed. With height-based scheduling, consumers are generally placed after the producers (except for operations on a recurrence cycle). Therefore, routing cost associated with just the producers is considered when an operation is placed. Even though the routing cost associated with consumers cannot be measured at the time that the producers are scheduled, it is desirable to account for these consumers in some way to avoid making greedy decisions. Ideally, operations with a common consumer should be placed close to each other so that the routing cost can be minimized later.

A measure of affinity is utilized to perform more intelligent scheduling by using information about common consumers. The affinity between a pair of operations with the same height is a measure of how close their common consumer is in the DFG. Operations with an immediate common consumer have the highest affinity be-



	Op 0	Op 1	Op 2	Op 3	Op 4	Op 5
Op 0	0	2	1	1	0	0
Op 1	2	0	1	1	0	0
Op 2	1	1	0	2	0	0
Op 3	1	1	2	0	0	0
Op 4	0	0	0	0	0	2
Op 5	0	0	0	0	2	0

Figure 4: Example affinity graph: (a) DFG for loop, (b) calculated affinities between each pair of operations, (c) affinity graph, and (d) possible operation assignments to a 2x4 CGRA.

tween them, while operations without a common consumer have zero affinity. Operations with indirect common consumers have moderate affinities that decrease based on the distance to the common consumer. The goal is to place operations with high affinity close together to minimize the routing cost of the common consumers.

For each pair of operations, the affinity is calculated by looking at their common consumers. An affinity graph is then constructed that consists of nodes representing operations and edges representing affinity between operations. An example of the affinity graph is shown in Figure 4. The affinity graph is constructed for the operations in the first row of the DFG in Figure 4(a). Figure 4(c) is the resulting affinity graph where solid edges represent high affinity between operations (a value of 2 in the example) and dotted edges represent low affinity between operations (a value of 1). Pairs of operations without edges have an affinity of zero.

For each pair of operations A and B with the same height, the affinity value is calculated using the following equation. Only the common consumers within the range of *max_dist* are considered in the calculation of the affinity value. The variable *num_cons(A,B,d)* denotes number of common consumers of A and B whose distance from A and B in the dataflow graph equals d.

$$affinity(A, B) = \sum_{d=1}^{max\_dist} 2^{max\_dist-d} \times num\_cons(A, B, d) \quad (1)$$

When scheduling operations, the scheduler attempts to place operations close together according to their affinity. Two alternate schedules for the operations are shown in Figure 4(d) that illustrate the use of affinity to eliminate explicit routing operations by performing more intelligent assignment of operations to nodes in the CGRA. The schedule on the left is better because operations with affinity edges are placed closer on the array.

3.1.5 Graph Embedding

In this work, we leverage graph embedding that is commonly used in graph layout and visualization. Graph embedding is a particular drawing of a graph onto a target space (usually a planar space). Drawing large graphs "nicely" is not an easy task. Here, a nice graph usually refers to non-crossing edges and a regular distribution of nodes. The spring embedder model [3] is a well known heuristic approach to graph embedding. It simulates a mechanical model of rings attached with springs. Each ring represents a node in the graph and each spring between two rings represents forces that attract or repel the nodes in the graph. The spring embedder is a suitable model for our scheduling. Each weighted edge in the affinity graph can be thought of as a spring that attracts two nodes in the graph. An edge with high affinity will attract two operations so that they are placed on the same or nearby resources.

A large amount of research has been conducted for effective graph drawing using the spring model. Kamada and Kawai proposed an iterative algorithm that calculates attractive and repulsive forces for each node and gradually moves the nodes with respect to the calculated forces [7]. Davidson and Harel employed a simulated annealing method that improves the cost of the graph based on the spring model [2]. However, most works do not fit into our scheduling problem as they assume a continuous space rather than the discrete, finite 3-D scheduling space. Graph embedding onto a grid-based space is well studied in the area of VLSI cell layout, known as force-directed placement. These works have somewhat different objectives, such as minimum edge bends. Li and Kurata proposed a grid layout algorithm of biochemical networks [12]. It uses simulated annealing for embedding complicated biochemical graphs onto the grid space. We found this solution best suited for our problem as its target space is discrete and the objective is placing nodes with edges close together.

Compared to the target graphs of typical graph embedding algorithms, our affinity graph has quite a small number of nodes. This is because we are not scheduling the whole application at one time. Instead, graph embedding is performed for each height level of the DFG and it is unusual for more than 20 operations to have the same height. Also, the search space is limited by pre-placed operations because pre-placed producers limit the possible slots of their consumers due to the sparse interconnect. For the search space that is sufficiently constrained, a simple exhaustive search can find an optimal layout of operations quickly. Li and Kurata's algorithm is employed only for large search spaces where the exhaustive search cannot finish in a reasonable time.

3.1.6 Skewed Scheduling Space

One of the difficult challenges of scheduling for CGRAs is ensuring that the necessary routing can take place as the CGRA is filled up with more operations. At the start of scheduling, the CGRA is empty, thus routing is not difficult. But, as scheduling proceeds, the scheduler can easily back itself into a corner and get stuck where the necessary routing cannot be performed. The affinity heuristic tries to minimize the overall number of resources used for routing, but this is not enough. When schedule times get larger than II, difficulties often result due to pre-placed operations (repeated resource use by the same operation every II cycles) and already-placed producers.

The conventional approach used in modulo scheduling is backtracking, where one or more operations are unscheduled to allow the current operation to successfully schedule [16]. However, backtracking for CGRAs is much more complicated. First, placing an operation usually requires more than one resource as routing is involved. This means that many operations can be unscheduled to overcome a routing failure. Moreover, re-scheduling operations requires both routing to its consumers as well as from its producers. It's difficult for the scheduler to make forward progress with backtracking.

A different approach is to prevent routing failures in advance. In general, routing failures to a consumer can be avoided if all the resources are free in time slots later than a producer's time slot. This is why the acyclic scheduling does not suffer from routing failures as it has an infinite scheduling space. Likewise, modulo scheduling does not suffer from routing failures within an II cycle window. Further, most applications don't have enough parallelism that re-

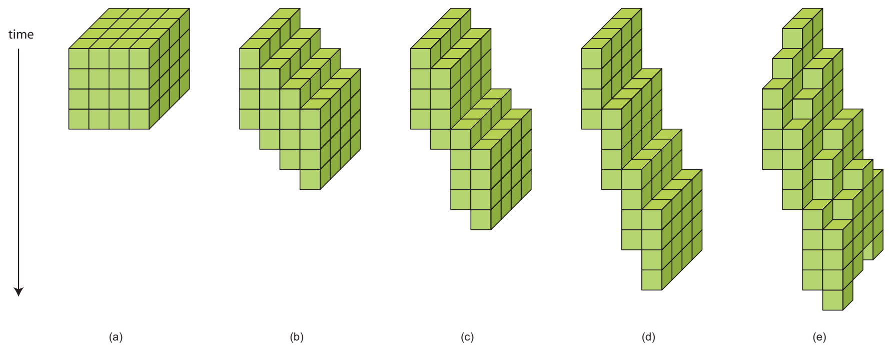


Figure 5: CGRA scheduling spaces: (a) normal scheduling space, (b) skewed scheduling space, (c,d,e) variations of skewed scheduling space.

quires all the CGRA FUs in one cycle. These two observations encourage clustering of the CGRA. With clustering, the FUs in the CGRA are partitioned into subsets. The scheduler can utilize one subset, or cluster, without any routing failures for II cycles. When the cluster is full, the scheduler can then use another cluster for the remaining operations, and so on.

Instead of partitioning the CGRA statically, our approach clusters the CGRA dynamically where the cluster boundaries are not strictly defined. The clusters are formed in a left-to-right manner on the array. The scheduler gives priority to the leftmost available FUs. But when the application parallelism is high, the cluster is dynamically enlarged by being forced to assign operations to lower priority FUs on the right. The scheduler utilizes a *position cost* to accomplish dynamic partitioning. When an operation is considered on an FU, its position cost is computed. The position cost is determined by the column in which the FU lies. Low cost is assigned to the leftmost available FUs, while higher cost is assigned to the FUs that lie further to the right.

When a partition of FUs to the left becomes full, values must be routed to FUs to the right. To guarantee this is possible, the concept of a skewed scheduling space is introduced as shown in Figure 5(b). Unlike the traditional scheduling space (see Figure 5(a)) where all the slots are available at the given schedule time, the start times of FUs are restricted such that they stagger down the right side of the CGRA. Since each FU is only available later than the FU on its left, the last schedule slot is always available to the output value of the last schedule slot of its left FU. When no operation is placed on an FU at the original start time, the start time increases, which slides down the scheduling space of the FU. When the scheduling space of an FU is lowered, scheduling spaces of FUs to its right are also lowered to guarantee the routability. Therefore, the skewed scheduling space dynamically changes as operations are placed in the CGRA. As the operations at the same height are considered together to get an optimal layout, the parallelism in the application at the given height determines the shape of the scheduling space. Some applications may not even require all four FUs in one column. In this case, the position cost is augmented with the row cost and the FUs in the upper rows are utilized first. Figure 5(c), (d) and (e) show several other possible skewed scheduling spaces.

Assignment of operations to the skewed scheduling space works well for forward dependence patterns, but difficulties arise with recurrence cycles. Recurrence cycles contain a communication pattern where a producer is scheduled after its consumer. Thus, the producer will be likely to be placed on the right of its consumer and routing becomes difficult since most schedule slots on the left are already utilized. To address this routing problem, our approach is to reserve in advance slots for such cycles when the consumer is placed. When a producer is placed later, it can use this reserved route. Again, we take the preventative approach to avoiding routing problems rather than solving them when they occur.

3.2 Implementation

Figure 6 presents an overview of our system. It takes the target loop body and description of the CGRA as input. The scheduling process consists of an initial preprocessing step to analyze the DFG and set up the skewed scheduling space. This is followed by the main scheduling loop that iterates over each level of the DFG to find a placement of all the operations at a particular height using modulo graph embedding.

3.2.1 Preprocessing

The target application is first preprocessed to calculate the heights of all operations based on the distance from the terminating operation (e.g., the loop back branch). The height of an operation determines when it is considered for scheduling and the height difference between producer and consumer is a rough estimation of the live range of the intermediate values.

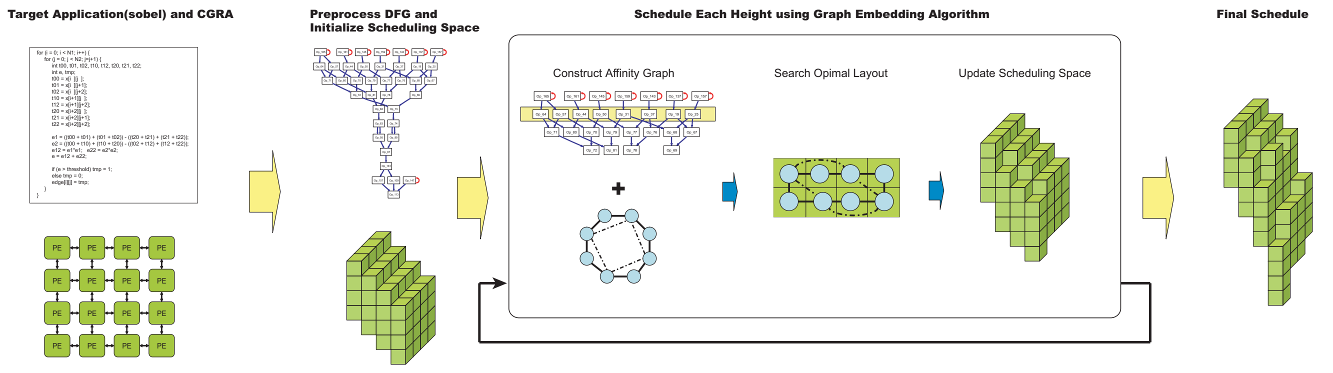
The scheduling space is skewed by assigning different start times to FUs. The same start time is assigned to all FUs in one column. Starting from zero for the first column on the left, the start time staggers downward with each increasing column number.

3.2.2 Scheduling Process

Scheduling proceeds through successive dependence height levels of the DFG considering all operations at a level simultaneously. Scheduling is converted into a graph embedding problem that maps the affinity graph onto the skewed scheduling space. Our modulo scheduler is implemented based on Li & Kurata's grid layout algorithm [12]. In the remainder of this section, we review basic ideas behind grid layout and describe our modified algorithm.

Grid Layout: Grid layout treats graph embedding as an optimization problem. A discrete cost function is defined for each pair of nodes based on the topological relation and the geometric positions of the nodes in the layout. Namely, high cost is given when two nodes connected by an edge are placed far apart and low cost is given when they are placed close together. The cost of a layout is given as a summation of costs for all node pairs. Simulated annealing is employed to find the layout with the lowest cost.

Modulo Graph Embedding: Unlike the original problem in grid layout, our problem has more constraints and costs to consider. Specifically, scheduling operations at each height has the following objectives:


Target Application(sobel) and CGRA
Preprocess DFG and Initialize Scheduling Space
Schedule Each Height using Graph Embedding Algorithm
Construct Affinity Graph
Search Opimal Layout
Update Scheduling Space
Final Schedule
PE

```
for (i = 0; i < N1; i++) {
    for (j = 0; j < N2; j=j+1) {
        int t00, t01, t02, t10, t12, t20, t21, t22;
        int e, tmp;
        t00 = x[i  ][j  ];
        t01 = x[i  ][j+1];
        t02 = x[i  ][j+2];
        t10 = x[i+1][j  ];
        t12 = x[i+1][j+2];
        t20 = x[i+2][j  ];
        t21 = x[i+2][j+1];
        t22 = x[i+2][j+2];

        e1 = ((t00 + t01) + (t01 + t02)) - ((t20 + t21) + (t21 + t22));
        e2 = ((t00 + t10) + (t10 + t20)) - ((t02 + t12) + (t12 + t22));
        e12 = e1*e1;   e22 = e2*e2;
        e = e12 + e22;

        if (e > threshold) tmp = 1;
        else tmp = 0;
        edge[i][j] = tmp;
    }
}
```

Figure 6: Overview of the CGRA scheduling system: input is the assembly code for the loop body and a description of the CGRA; preprocessing analyzes the loop to compute heights and skew the available scheduling cycles for the FUs; the graph is iterately scheduled at successive dependence height levels by constructing the affinity graph and performing modulo graph embedding of the affinity graph on the CGRA.

- Place operations with a common consumer close to each other
- Minimize the routing cost for values from producers
- Ensure the routability of values to consumers

To achieve the objectives above, the scheduling concepts in Section 3.1 are realized in a cost function composed of three terms: routing cost, affinity cost, and position cost. They are calculated for operations by the following equations. where A and B are operations to be placed and $affinity(A,B)$ is given by Equation 1 from Section 3.1.4:

$$routing\_cost(A) = \#\ FUs\ used\ for\ routing\ values\ from\ producers\ to\ A \quad (2)$$

$$affinity\_cost(A,B) = distance(FU(A), FU(B)) \times affinity(A,B) \quad (3)$$

$$position\_cost(A) = column\ \#\ of\ FU(A) \times BASE\_COST \quad (4)$$

$$layout\_cost = \sum_{A \in ops} \Big(routing\_cost(A) + position\_cost(A)\Big) + \sum_{A,B \in ops} affinity\_cost(A,B) \quad (5)$$

Grid layout employs a simulated annealing search to find an optimal layout of operations at each level by minimizing *layout_cost*. While the original grid layout maps a graph onto a 2-D plane, our target space is 3-D scheduling space which can have an infinite search space with varying schedule times. Therefore, we limit the search space by placing operations only in slots that minimize routing cost, called *primary slots*. Primary slots are identified before placing any operations. Even though each individual primary slot has the same routing cost, the total routing cost of a layout might vary because routing for one operation might block routing for another. Therefore, the routing cost is still considered in the cost function.

Once primary slots are identified, the size of search space is the product of the size of each operation's primary slots. Sometimes the search space can be quite small since pre-placed producers limit the placement of consumers. For small search spaces, exhaustive search is employed rather than using the grid layout. A flow chart of the scheduling process is presented in Figure 7.

The grid layout process begins with an initial layout obtained by randomly placing operations in one of their primary slots. Beginning with the initial layout, the scheduler enters a loop where the

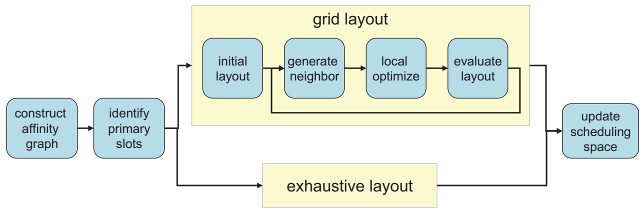


Figure 7: Scheduling process for operations at each successive dependence height.

cost of current layout is iteratively reduced using simulated annealing. First, operations are randomly moved or swapped with other operations to generate a neighbor of the current layout. The neighbor layout is then locally optimized. Local optimization greedily performs moving or swapping operations whenever the cost is reduced, and these actions are repeated until no further improvement can be achieved. The locally minimized layout is evaluated for acceptance as an optimal layout. At some points, an uphill movement is taken to escape from a local minima. After the optimal layout is discovered, the scheduling space is adjusted to reflect the chosen placement of operations at the current height and the scheduler proceeds to the next height.

3.2.3 Scheduling Example

The process of scheduling each height of the application onto the skewed scheduling space is illustrated in Figure 8 with sobel, an image edge detection algorithm. The II in this example is 4. Due to space limitations, scheduling of operations for the first three heights is presented. Figure 8(a) shows the DFG of sobel and the target 4x4 CGRA. Scheduling for the selected heights is illustrated in Figure 8(b).

For each height, the affinity graph for the operations is shown at the top with solid edges representing high affinity and dotted edges representing low affinity. The table in the middle, where FUs are represented horizontally and time vertically, shows the resulting layout of operations. Note that the FUs in the left two columns only appear in the table since the other FUs are not used in this example. Each entry in the table represents a schedule slot and shaded entries constitute the scheduling space of the CGRA (also

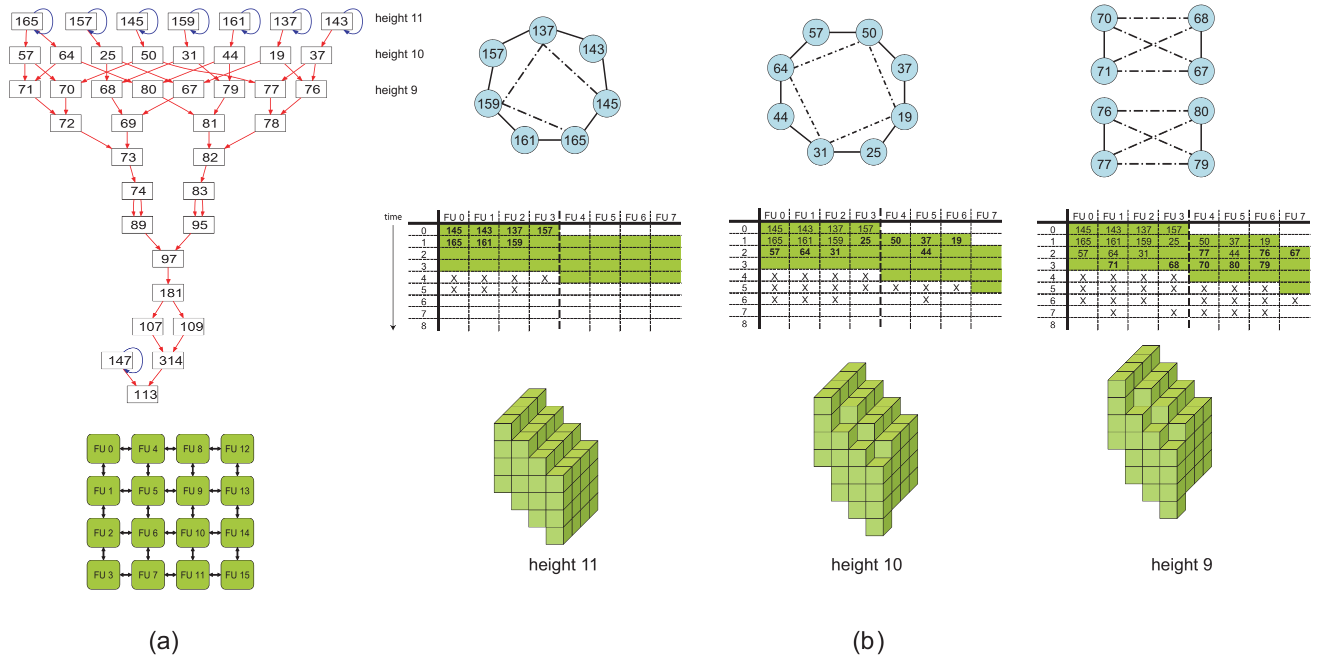


Figure 8: Example of modulo graph embedding: (a) DFG of sobel and target CGRA, (b) scheduling results of first three heights.

shown in 3-D graph at the bottom). Since FUs are repeatedly used every II cycles, entries are marked with X's when they are occupied by previously scheduled operations.

At height 11, operations are placed only in the first column due to the limit of the skewed scheduling space. Also, operations with high affinity represented in solid edges are placed in adjacent schedule slots. For example, 145 is placed adjacent to 165 and 143 due to its high affinity with these operations. Conversely, 145 is placed apart from 157 because there is no affinity between 145 and 157. Note that routing cost is not considered at this height since there are no producers placed.

At height 10, all the costs, including routing cost, are considered. As the operations at height 11 were intelligently placed based on the affinity, the scheduler places operations at height 10 without using any resources for routing. FUs in the second column are also utilized to support the parallelism in the application. Since no operation is placed on FU 7 at its original start time of 1, FU 7's start time is increased by 1 and its scheduling space is slid down. This also implies that the scheduling spaces of FU 11 and FU 15 are slid down to guarantee routability.

Operations at height 9 are scheduled similarly to those at height 10, again accounting for all costs. Note that the unoccupied slots in the second column at time 0 can be utilized II cycles later when output values of operations placed in the first column cannot otherwise be routed due to the modulo constraint. For example, the output values of operations 68 and 71 at height 9 can be routed using schedule slots of the second column at time 4.

The final scheduling space of sobel is shown on the righthand side of Figure 6.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

CGRAs can be characterized by many parameters. To evaluate the performance of our scheduler, three designs were tested.

Design Name	#RFs	#FUs per RF	#Regs	#Read ports	#Write ports
Dedicated RF	16	1	4	2	1
Shared RF	4	4	12	8	4
Central RF	16 local	1	4	2	1
	1 central	16	32	8	4

Table 1: Register file configurations for three CGRA designs used for evaluation.

All three designs have the same architectural parameters except for their register file configuration. All are 4x4 homogeneous CGRAs connected with a mesh network, with operation latencies of the ARM926 (e.g., 3 cycles for multiply, 2 cycles for load/store, and 1 cycle for simple arithmetic).

Table 1 shows the register file configurations for the three designs. These designs are the same as those pictured in Figure 1. The central RF design is the same as the dedicated RF design except that it has an additional central register file shared by all 16 FUs.

To evaluate the modulo graph embedding scheduler, twelve loop kernels are taken from various application domains: signal processing (fft, fir, iir, viterbi), encryption (blowfish), image processing (dct, fsed, sharp, sobel), network processing (channel), and video compression (idct, dequant). Only the innermost loop is considered for modulo scheduling for multidimensional loop nests.

4.2 Evaluation of Affinity Graph Heuristic

The main objective of the affinity graph heuristic is to minimize total routing cost by using common consumer information. In modulo scheduling, total routing cost is affected by other factors, such as recurrence cycles and the modulo constraint. In acyclic scheduling, we can exclude the influence of the modulo constraint as we can always find time slots where resources are available by increasing schedule time. Thus, we evaluated the performance of our affin-

	With Affinity		Without Affinity	
Bench	SchedLen	RouteFUs	SchedLen	RouteFUs
blowfish	32	4	34	14
channel	16	31	17	52
dct	15	24	19	53
fft	12	22	14	35
fir	8	3	9	5
fsed	11	2	12	6
sharp	21	19	25	23
sobel	11	2	13	12
viterbi	20	52	20	57

Table 2: Effectiveness of the affinity heuristic using acyclic scheduling.

ity graph heuristic in the domain of acyclic scheduling; only loop kernels without a constraining recurrence cycle were tested. The dedicated RF design in Figure 1(a) was used as the target architecture because it has the sparsest interconnect and therefore is the most affected by the placement heuristic.

Two cost models were compared to evaluate the affinity graph heuristic. One is implemented with both routing cost and affinity cost. (Position cost is not considered as the scheduling space is not skewed in this experiment.) The other model does not consider affinity in its cost function, and only tries to minimize routing cost when operations are placed. The quality of the schedule is measured by schedule length and number of FUs used for routing. The second and third columns of Table 2 show the quality of the schedules obtained with the affinity graph heuristic, while the fourth and fifth columns show the result without it. For all of the benchmarks, the affinity graph heuristic works well in reducing both the schedule length and number of FUs used for routing. Clearly, guiding placement using downstream information about consumers is important for CGRAs.

4.3 Evaluation of Modulo Scheduler

Two experiments are performed to evaluate the effectiveness of modulo graph embedding. First, a detailed analysis using the dedicated RF CGRA is presented. Then, we compare the most important parameter in scheduling for CGRAs, utilization or fraction of the cycles the FUs in the array perform useful computation, for all three register file configurations.

Scheduling results for the dedicated RF design are shown in Table 3. The second and third columns show the number of operations and the number of communication edges in the applications, respectively. These numbers roughly describe the communication patterns of the application. The fourth column shows the effective number of operations; for this metric, multi-cycle operations are counted multiple times according to their latency. Even if these operations can be pipelined with other operations of the same opcode, they increase the difficulty of the scheduling problem as the write-back resources of the node must be used at operation completion. The fifth and sixth columns in the table contain the minimum IIs for each benchmark. The maximum utilization that can be achieved is limited by these IIs.

The next two columns show the II and schedule length achieved by the modulo graph embedding scheduler. Unlike acyclic scheduling, II is a better measurement of performance than schedule length. The achieved II translates into the utilization of FUs shown in the "util" column. The utilization is calculated by dividing the number of schedule slots used for computation by the total number of slots which equals to (# FUs x II). "Total util," shown in the next column, takes into account the FUs being used for routing. All benchmarks show a utilization of greater than 43%. Fir has the lowest utilization, but more than half of the operations are multi-cycle operations, including four multiply operations. Iir also has low utilization, but its RecMII is 4, which limits the achievable utilization. Fft and sharp have relatively low utilization because they have a high number of one-to-many communication patterns. Routing cost increases with the number of consumers, as the value has to be individually routed to each consumer.

On average, the scheduler achieves 56% utilization for all benchmarks, with individual values ranging from 44% to 69%. This average utilization is similar to that achieved by the DRESC compiler, even though the target architecture of DRESC had a central register file and denser network connectivity. This shows that the modulo graph embedding scheduler is able to achieve quality solutions for significantly lower cost CGRAs.

The modulo scheduler runtimes (last column of Table 3) are reasonably fast, as all benchmarks are scheduled within 5 seconds on a 3 GHz Pentium-4 machine with 1G of RAM. This is because the search space is limited to operations in the DFG with the same height; thus, fewer than 20 operations are generally considered at a time. Also, scheduling does not employ backtracking, nor random movement of operations. Rather, systematic heuristics derived from the DFG guide the scheduler.

The impact of different register file configurations was evaluated by scheduling the same set of benchmarks on the other two CGRA designs (shared RF and central RF). The utilizations of the resulting schedules are shown in Figure 9. For all the benchmarks, the smallest II was achieved for the central RF, showing highest utilization in the graph except for blowfish and dequant. Blowfish and dequant were scheduled at the same II for shared RF and central RF, but utilizations are slightly higher for shared RF because different number of multi-cycle operations are pipelined. Since the central RF is connected to all 16 FUs, each FU can communicate with any other FU in 1 cycle, subject only to the availability of ports and register entries. With these additional routing resources, more FUs can be used for computation. The shared RF design achieves higher utilization than the dedicated RF design, as each register file can be used as a routing resource among the four FUs that share it. This result shows how increasing register file sharing can improve the quality of the schedule, giving more routing options to the scheduler.

5. RELATED WORK

5.1 Architectures

Many CGRA-like designs have been proposed in the literature. The designs have different scalability, performance, and compilability characteristics as discussed in Section 2.1. The ADRES architecture [14] is an example of an 8x8 mesh of processing elements with both individual and central register files. MorphoSys [13] is another example of an 8x8 grid with a more sophisticated interconnect network; each node contains an ALU and a small local register file. In the RAW architecture [18], each node is actually a MIPS processor, including memory, registers, and a processor pipeline. In addition, there are both dynamic and static routing networks. PipeRench [6] is a 1-D architecture in which processing elements are arranged in stripes to facilitate pipelining. RaPiD [4] consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network.

5.2 Compilation Techniques

Many techniques have been proposed for compiling to CGRAs. Lee et al. [9] propose a compilation approach for a generic CGRA.

Benchmark	# ops	# edges	eff # ops	ResMII	RecMII	II	sched len	util	total util	time (s)
blowfish	85	99	107	6	1	10	46	0.6500	0.8000	2
channel	121	180	187	8	1	16	30	0.6172	0.8789	4
dct	114	150	142	8	1	13	30	0.6250	0.8365	5
fft	52	78	78	4	1	9	25	0.4792	0.7986	1
fir	23	30	41	2	1	4	14	0.4375	0.7344	1
fsed	38	48	45	3	1	4	16	0.6875	0.9062	1
iir	23	33	32	2	4	4	15	0.4531	0.6250	1
sharp	56	95	72	4	4	9	37	0.4861	0.8542	1
sobel	39	59	52	3	1	5	17	0.6125	0.6750	1
viterbi	104	181	124	7	1	14	30	0.5268	0.8438	1
idct	119	200	215	8	2	18	41	0.5764	0.8333	5
dequant	84	141	106	6	3	10	25	0.6000	0.8063	1

Table 3: Modulo graph embedding results for the dedicated register file CGRA.

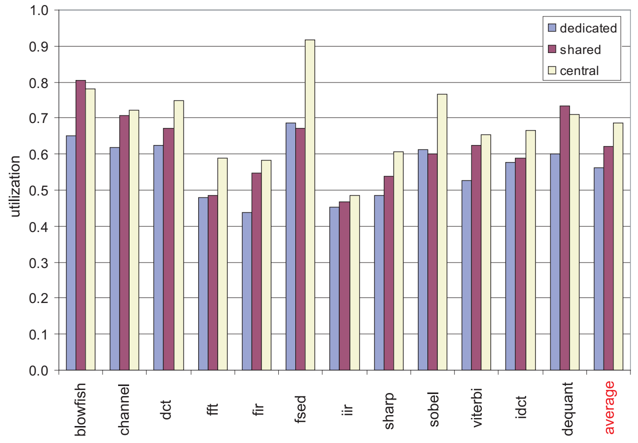


Figure 9: Comparison of utilization rates for three register file configurations.

They generate pipeline schedules for innermost loop bodies so that iterations can be issued successively. The main focus of their work is to enable memory sharing between operations of different iterations placed on the same processing element. Our work proposes a generic scheduling strategy, and memory sharing and other such optimizations can be integrated into our system as a preprocessing step. Convergent scheduling is proposed as a generic framework for instruction scheduling on spatial architectures [11]. Their framework comprises a series of heuristics that address independent concerns like load balancing, communication minimization, etc. Whereas convergent scheduling focuses on ILP and proposes a scheduling method for acyclic regions of code, we focus on loop level parallelism. The work of Mei et al. [14] is closest to our work. They propose a modulo scheduling algorithm for CGRAs based on simulated annealing. Our approach differs significantly in that we apply systematic placement decisions and on a skewed scheduling space to achieve better convergence and faster compilation times.

Similar to CGRAs, clustered VLIW machines are also spatial architectures. Much work has been done towards compiling for clustered VLIW machines [5, 15, 17]. Although some of the concepts from these works can be adapted for CGRA compilation, they do not consider the issue of routing values through the sparse interconnection network, which is a crucial step. The measure of affinity used in our scheduler is similar to that used in Krishnamurthy's affinity-based clustering [8].

[19] employs similar concept of affinity to minimize communication penalty in the resource allocation phase. A graph is constructed where nodes are operations and edges are inserted between nodes that have direct data dependences or common consumers. This graph is then partitioned into cliques and resource allocation is performed by assigning operations in each clique to the same resource. Time slots for operations are later assigned in scheduling phase. However, this approach that decouples resource allocation from scheduling is not suitable in modulo scheduling. Since each resource can be utilized only II times, it is not always possible to find proper time slots for operations on their pre-assigned resources. In our affinity graph heuristic, resource allocation is considered joinly with time slot assignment.

6. CONCLUSION

This paper proposes modulo graph embedding, an effective modulo scheduling technique for CGRAs. The sparse interconnect and distributed register files of the CGRA present difficult challenges to a compiler. Our approach leverages classic graph embedding to draw loop bodies onto a three dimensional graph representing the CGRA. We introduce two key concepts to generate high-quality solutions by reducing routing cost. First, an affinity graph heuristic analyzes producer/consumer relations to place operations with common consumers closely. Second, the scheduling space is skewed by restricting the assignable FUs and time slots available for each group of operations to enable dense packing of operations onto the array while still ensuring operand routing paths are available. Overall, modulo graph embedding achieves average compute utilizations of 56–68% for three different register file configurations, including a CGRA with no shared register files. Prior approaches have only achieved such utilization rates in CGRAs augmented with multiported shared register files. Our scheduler also performs substantially faster than existing solutions since we limit the search space to operations at the same height and employ a systematic placement based on the producer/consumer relations.

7. ACKNOWLEDGMENTS

Thanks to the anonymous referees who provided excellent feedback. This research was supported in part by ARM Limited, the National Science Foundation grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

8. REFERENCES

[1] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Apr. 2000.

[2] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[3] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[4] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997.

[5] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985.

[6] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999.

[7] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

[8] G. Krishnamurthy, E. Granston, and E. Stotzer. Affinity-based cluster assignment for unrolled loops. In *Proc. of the 2002 International Conference on Supercomputing*, pages 107–116, June 2002.

[9] J. Lee, K. Choi, and N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design & Test of Computers*, 20(1):26–33, Jan. 2003.

[10] W. Lee et al. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998.

[11] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 111–122, 2002.

[12] W. Li and H. Kurata. A grid layout algorithm for automatic drawing of biochemical networks. *Bioinformatics*, 21(9):2036–2042, 2005.

[13] G. Lu, H. Singh, M.-H. Lee, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference*, pages 727–734, 1999.

[14] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the 2003 Design, Automation and Test in Europe*, pages 296–301, 2003.

[15] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1998.

[16] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[17] J. Sanchez and A. Gonzalez. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 124–133, Dec. 2000.

[18] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[19] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. pages 116–125, 2001.