Mapping loops onto Coarse-Grained Reconfigurable Architectures using Particle Swarm Optimization

Rani Gnanaolivu, Theodore S. Norvell, Ramachandran Venkatesan
Faculty of Electrical and Computer Engineering
Memorial University of Newfoundland
St. John's, NL, Canada A1B 3X5
{ranig, theo, venky}@mun.ca

***Abstract*— Coarse-Grained Reconfigurable Architectures (CGRAs) have gained currency in recent years due to their abundant parallelism and flexibility. To utilize the abundant parallelism found in CGRAs, we propose a fast and efficient Modulo-Constrained Hybrid Particle Swarm Optimization (MCHPSO) scheduling algorithm to exploit loop level parallelism in applications. PSO has been proved to be successful in many applications in continuous optimization problems. In this paper, we show that PSO is capable of software pipelining loops by overlapping placement, scheduling and routing of successive loop iterations and executing them in parallel. Our proposed algorithm has been experimentally validated on various DSP benchmarks under two different architecture configurations. These experiments indicate that the proposed MCHPSO algorithm can find schedules with small initiation intervals within a reasonable amount of time. PSO is thus a promising alternative for obtaining near optimal solutions to this NP-hard scheduling problem.**

Keywords- Coarse-Grained Reconfigurable Architectures; Particle Swarm Optimization; Modulo Scheduling; Loop level parallelism; Mapping.

I. INTRODUCTION

Reconfigurable Systems have drawn increasing attention from both academic and commercial research applications in the past few years because they combine flexibility with efficiency and upgradability [*1*]. Among the reconfigurable architectures, many Coarse-Grained Reconfigurable Architectures (GGRAs) have been proposed as an alternative to FPGA-based systems [*2*]. CGRAs consist of programmable coarse-grained Processing Elements (PEs) which support a predefined set of word-level operations, a programmable interconnection network, a configuration memory, and a controller [*1*]. Unfortunately the available parallelism has been exploited by few automated design and compilation tools [*2*].

The massive amounts of parallelism found in CGRAs can be used to map time critical loops of an application. This can be achieved by Modulo Scheduling [*2*], which is a software pipelining technique that overlaps several iterations of a loop by generating a schedule for an iteration of the loop. Modulo scheduling uses the same schedule for subsequent iterations started at a constant interval called the initiation interval (II).

Several heuristic techniques have been tried by researchers in solving the modulo scheduling problem. In this paper, we propose a modulo scheduling algorithm based on Particle Swarm Optimization (PSO). We call this the Modulo-Constrained Hybrid Particle Swarm Optimization (MCHPSO) algorithm. PSO provides a near optimal solution with fast convergence and low execution time in solving various combinatory and multidimensional space optimization problems [*3*]. The MCHPSO algorithm enforces modulo constraints on the parallelism of loop operations as well as data dependence, while mapping onto the CGRA.

The MCHPSO algorithm has been tested on benchmarks taken from [*4*], [*5*], and [*6*]. The benchmarks are derived from applications written in the C programming language. The results show that the proposed MCHPSO algorithm finds a valid schedule for the given target applications in reasonable time, with efficient utilization of resources.

The rest of this paper is organized as follows: An overview of compilation and background is given in Section II. The proposed PSO-based modulo scheduling algorithm (MCHPSO) is explained in Section III. The last three sections present the experimental results, related work, and conclusion.

II. BACKGROUND

In this paper, we propose an algorithm for modulo scheduling of a loop to be mapped onto CGRAs. The method starts from an imperative language representation of the application, such as a program written in C or some other high-level language.

Each source program is converted to a Data Flow Graph (DFG). The given Target Architecture (TA) is represented by a graph containing all the necessary information such as the number of resources, capacity and interconnections as well as other specific information for each resource. The generic TA graph representation was designed to allow a wide range of architectures. The ADRES [*1*] architecture was adopted as the TA for our current work. We chose ADRES architecture because it has a flexible architecture template and we can easily map loops onto the ADRES array in a highly parallel way. Furthermore, choosing this architecture allows direct comparison with the method presented in [*1*]. The TA is replicated for each time cycle to form the Routing Resource Graph (RRG), an internal time-space graph representation.

The mapping algorithm MCHPSO maps each node of the DFG to a node of the RRG and each edge of the DFG to a path in the RRG. The generated scheduled code of the loop exhibits a high degree of Instruction Level Parallelism (ILP).

A. Motivational Example

The compilation flow with a motivational example is described in Figure 1. Consider the architecture configuration

978-1-4244-7896-5/10/$26.00 ©2010 IEEE

taken in Figure 1 (a), and a DFG represented in Figure 1 (c). The architecture components in Figure 1 (a) are Input port (I), Functional Unit (FU), Write Port (WP), Read Port (RP), Register File (RF). Figure 1 (b) shows an RRG created by replicating the TA across two time cycles, as the II is 2. The final embedding of DFG on RRG is shown in Figure 1 (d).

The schedule produced by the algorithm maps each operation to a processing element and a time and maps each edge in the DFG to a path in the RRG. During the scheduling process, the MCHPSO keeps track of the resources being used in a Modulo Reservation Table (MRT), as shown in TABLE I.

The columns in the MRT represent the resources in the architecture and the rows represent remainders modulo the initiation interval. The operation $n1$ is to be executed in $FU1$ at time 0, so the FU1 is reserved for all cycles divisible by II. Once a resource is reserved it will not be available for the other operations in time cycles that have the same remainder modulo *II*. The routing path from operation $n1$ to operation $n2$ uses the $\mathrm{WP}_{1\ (2)}$, $\mathrm{RF}_{1\ (2)}$, $\mathrm{RP}_{1\ (2)}$, which are also reserved in the MRT. The capacity of resources are given in brackets, otherwise they have capacity of one.

B. *Particle Swarm Optimization*

Particle Swarm Optimization (PSO) is an optimization approach that follows an evolutionary metaphor. It is a population-based search procedure in which individuals, called particles, changes their positions, or states, with time. Each particle in the PSO system represents a potential solution to the problem, and at the end of the search, the best particle will hold the best solution found. The standard PSO is discussed in [*3*].

In every iteration, the velocity and position of each particle are calculated according to the expressions given below.

$$V_{i+1} := w \times V_i + c_1 r_1 (L_i - X_i) \quad + c_2 r_2 (G_i - X_i) \qquad (1)$$

$$X_{i+1} := X_i + V_{i+1} \qquad (2)$$

where

$$w = w_{max} - \frac{w_{max} - w_{min}}{iter_{max}} \times i \qquad (3)$$

i denotes the current iteration and $iter_{max}$ the maximum number of iterations. X_i denotes the particle coordinates at i $V_i$denotes the velocity at$i$. c_1 ,and c_2 denote the acceleration constants in the range $[0, 1]$ and r_1 and r_2 are random values in the range$[0, 1]$. L_i and $G_i$denote the local best particle position and global best particle position at the i^{th} iteration. w denotes the inertia weight factor with w_{min}, w_{max} as the initial weight and final weight.

After calculating X_{i+1}, we can get the new particle position to search in the next iteration. PSO algorithm has the advantages of high speed, stable convergence and robustness; it is parallelized well and generates good solutions [*3*].

PSO shows significant performance in the initial iterations when compared with Ant Colony Optimization (ACO). PSO has the capability to quickly arrive at an optimal/near-optimal solution [7]. An advantage of PSO over Genetic Algorithm (GA) is that PSO maintains all the solutions in the search space and changes of inertia weight leads to convergence [8]. Since previous research on PSO [3], [9] shows that scheduling can be done with PSO, we tried PSO with a hybrid combination of mutation operations for our Modulo Scheduling problem to avoid premature convergence in PSO algorithm.

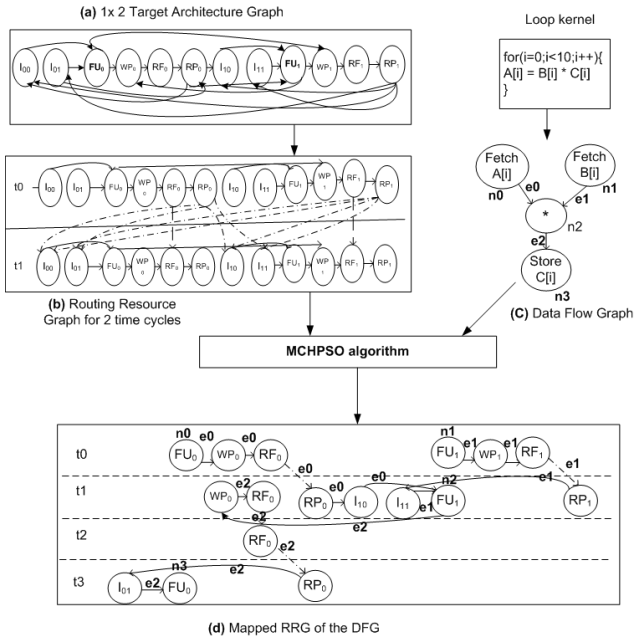

Figure 1. Motivating example a) 2 x 2 target architecture template instance, b) RRG, c) DFG andd) Final schedule, place and route result.

TABLE I. MRT FOR THE DFG INFIGURE 1 (C)

resource / II	I_{00}	I_{01}	FU_0	$WP_{0\ (2)}$	$RF_{0\ (2)}$	$RP_{0\ (2)}$	I_{10}	I_{11}	FU_1	$WP_{1\ (2)}$	$RF_{1\ (2)}$	$RP_{1\ (2)}$
0			n0	e0	e0,e2				n1	e1	e1	
1		e2	n3	e2	e2	e0,e2	e0	e1	n2			e1

C. *Target Architecture Graph*

The target architecture consists of a graph of basic components, including Functional Units (FUs), Register Files (RFs), Column Buses (CBs), and Row Buses (RBs). Similar to the work done in [*2*] and [*10*], our work aims to target a wide range of CGRAs. For the experiments reported in Section V, we targeted an architecture similar to the ADRES [*1*] architecture template.

The TA graph (V, E) is formed from a target description file where,

- V is the set of vertices. Each vertex represents a FU or RF or CB or RBs described above.
- E is the set of edges, indicating the incoming or outgoing edge in the operation. $\overleftarrow{e}$ and $\vec{e}$ are the source and target vertex for edge e.

Each FU can receive input from various resources of the graph and similarly the output of each FU can be routed to various destination resources [*1*]. The target architecture used in the experiments of Section V has both 4x4 instances and 8x8 instances of FUs. An example 4x4 instance of target architecture is shown in Figure 2. Only the top row of FUs, termed as Memory Unit (MU), may be used for load and store operations.

D. Routing Resource Graph

For scheduling, placing, and routing loops onto the target architecture, we employ a time-space graph called a Routing Resource Graph (RRG). The RRG is obtained from the TA graph described above by replicating each vertex in V for every time cycle $\epsilon\mathbb{N}$ specifyingthe interconnections with edges derived from$E$. The RRG is $(V \times \mathbb{N}, X \cup Y \cup Z)$ where

- $V \times \mathbb{N}$ – An infinite set of copies of the TA's vertex set.

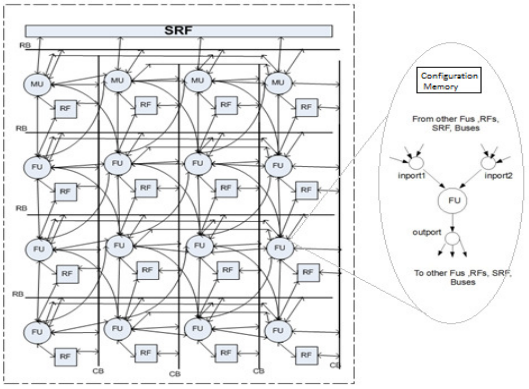

Figure 2. 4 x 4 target architecture template instance.

- X edges – Every incoming edge e in the TA graph that doesn't end at a register write port is replicated across time.
- Yedges – Every incoming edge e in the TA graph that ends at a register write port is represented in the RRG as an outgoing edge from its source in current time cycle to the write port in the next time cycle. Use of such an edge represents writing to a register [*11*].
- Zedges – For every RF r in the TA graph, we have a set of edges that transmit data from each instance of the RF to the instance in the next time. Use of such an edge represents maintaining data in a register [*11*].

E. Initiation Interval

To enforce the modulo constraints, we have to generate a schedule for one iteration of the loop, such that this same schedule is repeated at regular intervals with respect to data dependences and resource constraints [1]. This interval is termed the Initiation Interval (II), essentially reflecting the performance of the scheduled loop. To start the MCHPSO scheduling process, the II is assigned the value of a lower bound called as Minimum Initiation Interval (MII) and is computed as in [*1*].

F. Data Flow Graph

The target application program description is analyzed and transformed to find the critical loops to be mapped to the CGRA. In our work, we have considered only the inner loop body of the application with no inter-iteration dependence. The loop kernel is rewritten to create a data flow graph representation with nodes as the set of operations in the loop kernel and arcs as the set of interconnection edges, indicating the incoming or outgoing edge of the operation [7].

III. MAPPING ALGORITHM

A. Modulo Scheduling

Modulo Scheduling is a technique for software pipelining loops [*2*]. The schedule for each iteration is divided into stages of equal duration, so that different stages of the successive iterations get overlapped. The number of stages in each iteration is called the Stage Count (SC). Modulo scheduling ensures that there are no resource conflicts as multiple stages execute simultaneously.

B. Proposed Algorithm

1) Modulo scheduling with Modulo Constrained Hybrid Particle Swarm Optimization

Our proposed MCHPSO scheduling algorithm simultaneously searches for a good schedule, placement, and routing solution for the entire set of operations given in DFG; it avoids the time consuming sequential search for each operation proposed in the mapping algorithm described in [*2*]. In [*2*], [*11*], [*10*] several trials are needed to find the best schedule for an operation before proceeding to the next operation. In our algorithm, all the particles search for a complete schedule simultaneously. To efficiently map loops onto the CGRA, we have adopted the idea of modulo scheduling used in [*2*] along with the combination of two heuristic approaches, PSO and randomization. From [*3*] and [*9*] we note that PSO could be applied to multidimensional scheduling problems. The application of PSO to modulo scheduling converges faster but can be caught in a local optimum. To escape the local optima, we have used a randomization method in combination with PSO. The overall method of MCHPSO to schedule, place and route a loop is explained in Figure 3. The inputs to the algorithm are TA graph and a DFG.

```
Procedure ModuloSch_Place_Route (DFG, TA)
begin
  II := MII (DFG)
  dfgList := ComputeASAPandALAP (DFG)
  sortedDFG := sort(dfglist)
  max_schLength := findschLength(sortDFG)
  schSucess := false
  trials :=0
  while !schSucess&& trials<NTRIALS do
    CreateRRG(TA, II, max_schLength)
    schSucess:=MCHPSO(sortedDFG, RRG, II, max_schLength)
     II++
     trials++
  end while
end
```

Figure 3. Mapping DFG toRRG

First the Minimum Initiation Interval (MII) is computed as discussed in the previous section. Second, ASAP (As Soon As Possible) and ALAP (As Late As Possible) times are calculated as in [*2*] for the given DFG. After generating the DFG and the RRG, the MCHPSO algorithm is executed to schedule, place, and route the loop.

2) Particle Encoding for the problem

To frame the solution for the scheduling problem by using the particles, we need to consider various dimensions for each particle, size of DFG, placement of nodes, routing and the schedule time. To establish "best solution mapping", we have

taken each particle position as a mapping of DFG nodes to RRG nodes and DFG edges to RRG paths.

3) MCHPSO

In MCHPSO, inputs are the RRG and the sorted DFG. The number of operations in the DFG is initialized to the number of nodes, *N*, for each particle. Each particle in the PSO takes the initial value for the place and schedule of each node in the range of [ASAP, ALAP] that satisfies the dependence constraint. Once all the particles are initialized, their fitness is calculated as illustrated in the next subsection. Every particle updates its Local-best ($Lbest$) position if the new fitness is better than the current fitness. Once all the particles have been updated to their best candidate solution, the global best particle is chosen and its position is denoted by $PGbest$ the global best particle is chosen and its position is denoted by $PGbest$.

Every particle i updates its velocity according to (4). The $createSwapList$ function in (4) creates a swap sequence [*3*] of the current particle's ($currentP_i$) placed and scheduled nodes with either from global best position (P_{Gbest}) or from the local best position (P_{Lbest}). Once the new velocity ($Vnew_i$) is generated, the current particle position ($currentP_i$) is swapped according to the co-ordinates in the $Vnew_i$ as in (5). Next the mutation operator is applied to the new particle position ($newPcoord_i$) is shown in (6). The $mutationOperator$ function selects a random node of the particle and chooses a random placement and schedule value and replaces the particle's current value. Once the mutation is done on the particle, the new particle coordinates are ready for the next generation of MCHPSO. The particles keep searching for the best solution in the current II. The pseudo code is shown in Figure 4.

4) Fitness calculation

The fitness calculation considers multiple objectives from the routing path produced by Dijkstra's shortest-path algorithm [*12*]. The three main objectives considered in our work are that no resource is overused, that all edges in the DFG are routable, and that few resources are used to route. The routing cost is computed by accumulating the cost of all RRG nodes used by the new placement and routing of the operation. The fitness calculation was designed to penalize particles which overuse resources. Each node in the RRG has a capacity, base cost [*2*], availability, and usage number. The majority of RRG nodes have a capacity of one whereas a few types of nodes such as register files have a capacity larger than one.

$$Vnew_i := \textbf{if } C_1 \textbf{ then } CreateSwapList(P_{Gbest}, currentP_i) \textbf{ else } CreateSwapList(P_{Lbest}, currentP_i) \quad (4)$$

where C_1 is an acceleration constant ranges $[0, 2]$.

$$newPcoord_i := doSwap(currentP_i, Vnewi) \quad (5)$$

$$newPcoord_i := mutationOperator(newPcoord_i) \quad (6)$$

```
Procedure MCHPSO (sortDFG, RRG, II, schLength)
begin
for each operation in sortDFGdo
        Initialize Particles
        InitializeMRT(noofFU,II)
end for
repeat NLOOPS times
        for each particle in Particles do
          Find the fitness value fromGetRoutingCost (RRG, particle)
          if the fitness value is better than the best fitnessthen
              Set current fitness value as the new particle best fitness
          end if
        end for
        Find the global best particle
        for each particle do
            Calculate the new particle velocity according to (4)
            Update particle search position according to (5)
            Apply mutation operator for the newPosition (6)
        end for
end while
if validSchedule(bestparticle) then return true
else return false
endif
end
```

Figure 4. The MCHPSO algorithm

IV. EXPERIMENT

A. Set up

The proposed scheduling algorithm was written in Java and executed on an Intel Core 2 Duo CPU with 4 GB RAM and a clock speed of 2 GHz. To schedule a loop onto the CGRAs, two main inputs were required for the scheduling algorithm. The first input is the DFG generated from the benchmark loops. The second input for the MCHPSO is the CGRA configuration. The TA graph is created from the TA configuration.

Other than the two main inputs, DFG and TA, MCHPSO requires the following parameters: the number of particles is 10, the relax-factor for the schedule length is the II of the DFG, C_1 as one or zero depending on the random generation, the number of trials for each II is one, and the number of iterations to carry out the algorithm is 20.

Among the various CGRAs discussed in [*1*], Architecture for Dynamically Reconfigurable Embedded Systems (ADRES) [*2*] was used for the experiments. The TA consists of 64 FUs, which are divided into four tiles. Each tile consists of 16 FUs in a 4 by 4 grid as shown in Figure 2. The benchmarks used consist of ten programs, which are derived from [*4*], [*5*], and [*6*].

B. Experiment Results

The overall mapping results of all the selected benchmarks are shown in TABLE II where the first column shows the benchmark name, second column denotes the number of operations in the loop kernel, and the third column shows the Initiation Interval (II) at which the loop kernel is mapped. The fourth column shows the Operations Per Cycle (OPC) which is calculated by (7). The fifth column shows the schedule density without routing, calculated as in (8). The schedule density without routing considers the count of FUs used in the placement. The sixth column shows the schedule density of FU with routing calculated by (9), where the number of stages

is calculated by (10). The schedule density with routing considers the count of FUs used in the placement as well as in routing of edges. The seventh column shows the total CGRA utilization percentage, including all the computation and routing resources in the CGRA used for the scheduling of loop kernel calculated by (11).

$$OPC = \frac{N_{Oerations}}{II} \tag{7}$$

$$\text{Schedule density without routing} = \left(\frac{OPC}{number\ of\ FU}\right) * 100 \tag{8}$$

$$\text{Schedule density with routing} = noofstages * \left(\frac{N_{Operations} + FUusedinrouting}{\text{number of FU}}\right) * 100 \tag{9}$$

The eighth column shows the number of stages overlapped, as calculated in (10). The last column shows the time taken in seconds to schedule the loop kernel. The mapping results show that the proposed scheduling algorithm MCHPSO utilizes from 31.25% to 79.69% of the total FUs available in the CGRA. The FU utilization depends on the size of the DFG and the number of stages through which a loop is unrolled. The largest loop kernels like IDCT_hor (horizontal pass) and FFT are scheduled within a maximum of 105.89 seconds.

The usage of Functional Units in the CGRA instance has been studied in Figure 5. From the mapping results, it is understood that the higher the number of loop operations, the larger the routing resources required.

$$number\ of\ stages = \left\lceil \frac{Schedule\ Length}{II} \right\rceil \tag{10}$$

$$TotalUtilization = numberofstages * \left(\frac{N_{Operations} + total_{routingRes}}{RRGsize}\right) * 100 \tag{11}$$

C. Comparison of MCHPSO with other modulo scheduling algorithms

TABLE III shows the comparative results of MCHPSO measured against the modulo scheduling algorithm [*1*] used in ADRES architecture. The first column shows the benchmarks taken for comparison. The second and seventh columns show the number of operations derived from the benchmarks on both the algorithms.

The third and eighth columns show the II at which both the algorithm were able to do the loop level parallelism. The fourth and ninth columns show the schedule density of FU (with routing). The fifth and tenth columns show the Operations Per Cycle (OPC) as calculated in (7). The sixth and eleventh columns show the scheduling time in seconds for the mapping of the benchmark. The comparison shows that our proposed MCHPSO algorithm was able to route the FFT benchmark within the minimum II with a small measure of execution time.

TABLE IV shows the comparison of MCHPSO with the modulo scheduling algorithm used in [*10*]. The authors of this paper have used a 2D CGRA with 16 PE with PEIT1 (all PEs are connected with its row PEs and column PEs) and PEIT2 (nearest neighbour) topology. The execution time is smaller in the PEIT1 than in PEIT2 because there is a smaller average routing delay experienced by PEIT2 while PEIT1 overcomes the routing delay by the richer interconnection topology. A memory-conscious mapping algorithm based on the priority-based list scheduling algorithm is used in [*10*]. Therefore, we have compared the work done in [*10*] based on PEIT1 with our proposed algorithm. The first column in TABLE III shows the benchmarks taken for comparison. The second and sixth columns show the number of operations in the benchmark. The third and seventh column shows the Operations Per Cycle (OPC) as calculated in (7) and the fifth and ninth columns show the schedule density of FU (with routing) as calculated in (9).

This comparative study has established that our proposed algorithm has a lower schedule density (with routing) and minimal II for the first four benchmarks in spite of not using L1 and L0 scratch pad memory, which has been used in [*10*]. The fifth benchmark 8x8 IDCT-hor depicts a typical case of showing that our algorithm maps at a lower II with the same number of operations and schedule density compared with results in [*10*]. The numbers of operations are different for the comparing algorithms, because of the various analysis and transformation phase carried out in [1] and [11]. Our proposed algorithm achieves to map with a minimal II for all the benchmarks taken for comparison to the work done in [11] with better utilization of resources. Our proposed algorithm achieves to map with a minimal II for all the benchmarks taken for comparison to the work done in [11] with better utilization of resources.

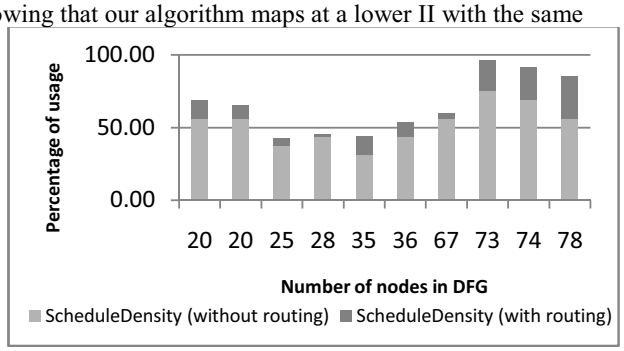

Figure 1. Shows the scheduling density usage percentage with and without routing in a 4 x 4 CGRA instance for the DFGs.

V. RELATED WORK

In the CGRA compilation, Software Pipelining [*13*] is used for instruction parallelism. The idea of software pipelining is to look for a pattern of operations from various iterations (often termed as the kernel) so that when repeatedly iterating over this pattern, it produces the effect that next iteration is initiated at a regular interval. This interval is termed the Initiation Interval (II) which essentially reflects when the next iteration can start to increase the performance of the scheduled loop. Some of the approaches carried out in modulo scheduling of the inner loop body are discussed below.

The compilation of inner loop body in CGRAs has been done with DRESC (Dynamically Reconfigurable Embedded System Compiler) [*2*], a retargetable compiler that is able to parse, analyze, place, route, and schedule the C source code. In this work they propose a modulo scheduling algorithm based on simulated annealing where it takes a long compilation time for larger loops.

A memory-conscious mapping methodology for CGRA architectures was presented in [*10*] with data reuse capabilities and priority-based list scheduling algorithm. The resource aware mapping with local RAMs and flexible interconnection network enables to map the application. The idea of modulo scheduling is applied with a graph embedding technique using an affinity graph heuristic and skewed scheduling space in [*14*]. The method achieves better convergence and faster compilation times with dedicated register files and sparse network connectivity.

The discrete problem of Instruction scheduling has been solved using Particle Swarm Optimization PSO with the traditional list scheduling algorithm [*3*]. Our approach closely resembles the work in [*2*] and [*3*] by using hybrid PSO with mutation operator to decide the placement and scheduling decisions in CGRAs. The routing path value for the fitness function is calculated from Dijikstra's algorithm to achieve better convergence and faster compilation times. In contrast to all the algorithms discussed, our approach takes the evolutionary process to decide the simultaneous mapping decisions for all the nodes in the DFG. The proposed algorithm optimizes the routing cost as well as holds the modulo constraints and data dependence.

TABLE I. MCHPSO -- OVERALL MAPPING RESULTS FOR 8 X 8 CGRA

Bench-Marks	# of ops	II	OPC	Schedule Density (without routing)	Schedule Density (with routing)	Total CGRA Util %	No of stages	Exe Time in Seconds
FIR_complex	25	2	12.5	18.75	39.06	12.59	4.00	8.72
Lattice synth	20	1	20.0	29.69	79.69	22.06	10.00	12.58
Volterra	28	2	14.0	21.88	34.38	14.06	3.00	6.87
IIR	36	2	18.0	28.13	62.50	21.14	4.00	12.55
IIR_biquad	35	3	11.7	17.19	31.25	9.25	4.00	16.93
8X8 IDCT_hor	78	3	26.0	40.63	73.44	29.47	5.00	93.11
4X4 FFT	67	3	22.3	34.38	75.52	29.66	5.00	105.89
8X8 FDCT_hor	74	4	18.5	29.69	63.28	18.34	3.00	27.01
8X8 FDCT_Ver	73	3	24.3	37.50	78.13	21.20	4.00	55.67

TABLE II. COMPARISON OF MCHPSO WITH RESULTS IN [*1*]

Comparing algorithms	8 x 8 MCHPSO					Results reported in [*1*]				
Benchmarks	*# of ops*	*II*	*Schedule Density (with routing)*	*OPC*	*Exe Time in Seconds*	*# of ops*	*II*	*Schedule Density (with routing)*	*OPC*	*Exe Time in Seconds*
8X8 IDCT_hor	78	3	73.44	26.00	93.11	128	3	90.10%	42.70	340
4X4 FFT	67	3	75.52	24.00	105.89	79	4	75.00%	19.80	314

TABLE III. COMPARISON OF MCHPSO WITH RESULTS IN [*10*]

Comparing algorithms	**4 X 4 MCHPSO**				**Results reported in [*10*]**			
Benchmarks	***# of Ops***	***II***	***OPC***	***Schedule Density (with routing)***	***# of Ops***	***II***	***OPC***	***Schedule Density (with routing)***
latasynth	20	2	9.0	68.75	18	6	3.0	75.00
Volterra	28	4	7.0	45.31	27	7	3.9	70.30
IIR	36	5	7.0	43.75	39	8	4.9	59.50
4X4 FFT	67	7	9.0	59.82	95	17	5.6	69.60
8X8 IDCT_hor	78	6	13.0	85.16	79	14	5.6	85.10
latanal	20	2	9.0	65.63	18	8	2.3	62.50

VI. CONCLUSION AND FUTURE WORK

In this paper, we have proposed the Modulo Constrained Hybrid Particle Swarm Optimization (MCHPSO) algorithm for the loop scheduling problem in CGRAs. The results from our proposed algorithm indicate that the algorithm can find a valid schedule, placement and routing for the given benchmark loops on required initiation interval and maps with a good utilization of resources. Our algorithm can be enhanced to exploit if-conversion, conditional branches and inter-iteration dependence in the loop exploitation. In our future work, we will be trying to apply the proposed algorithm on various reconfigurable architectures and complex applications. The results produced by MCHPSO will be compared with other hybrid evolutionary algorithms in the future. To study the parallelization of the mapping solution search in the proposed algorithm, we have tried on a quad core machine with eight logical processors. The preliminary results are promising and will be discussed in the future paper.

REFERENCES

[1] B. Mei, M. Berekovic, and J.-Y. Mignolet, Fine- and Coarse-Grain Reconfigurable Computing. Springer Netherlands, 2007, ch. 6, pp. 255-297.

[2] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," Computers and Digital Techniques, IEE Proceedings, vol. 150, no. 5, pp. 255–261, Sept. 2003.

[3] Rehab F. Abdel-Kader, "Particle Swarm Optimization for Constrained Instruction Scheduling," VLSI Design, vol. 2008, Article ID 930610, 7 pages, 2008. doi:10.1155/2008/930610

[4] Texas Instruments. inc, "DSP Benchmarks," http://dspvillage.ti.com, May 2009.

[5] Texas A&M University-Kingsville, "Lattice LPC analysis filter."http://www.engineer.tamuk.edu/SPark/Analysis-Synthesis.htm, December 2009.

[6] University of Patras, "VLSI design."http://www.vlsi.ee.upatras.gr, December 2009.

[7] S.Nonsiri and S.Supratid, "Modifying Ant Colony Optimization," Soft Computing in Industrial Applications, 2008. SMCia '08. IEEE Conference on , vol., no., pp.95-100, 25-27 June 2008.

[8] A. Chatterjee and P. Siarry, "Nonlinear inertia weight variation for dynamic adaptation in particle swarm optimization," Computers and Operations Research, Volume 33, Issue 3, March 2006, Pages 859-871.

[9] T.Chiang, P. Chang, and Y.Huang, "Multi-Processor Tasks with Resource and Timing Constraints Using Particle Swarm Optimization," *IJCSNS International Journal of Computer Science and Network Security*, vol. 6, 2006.

[10] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis, "Design space exploration of an optimized compiler approach for a generic reconfigurable array architecture," *Journal of Supercomputing*, vol. 40, pp.127-157,2007.

[11] M. Tuhin and T. Norvell, "Compiling parallel applications to coarse-grained reconfigurable architectures," in Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on, May 2008, pp. 001 723 –001 728.

[12] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[13] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software Pipelining," ACM Comput. Surv., vol. 27, no. 3, pp. 367-432, 1995.

[14] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures," in CASES Š06: Proceedings of the 2006 international conference on Compilers, architecture, pp. 136-146, ACM Press, 2006.