Published in IJoOCI vol. 2 no. 2, 2011 (c) IJoOCI

Analysis of inner-loop mapping onto Coarse-Grained Reconfigurable Architectures using Hybrid Particle Swarm Optimization

Rani Gnanaolivu, Theodore S. Norvell, Ramachandran Venkatesan

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 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. In this paper, we show that Particle Swarm Optimization (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. The MCHPSO scheduling algorithm was analyzed with different topologies and Functional Unit (FU) configurations. We tested the parallelizability of the algorithm and found that it has nearly linear speedup on a multi-core CPU.

Keywords: Coarse-Grained Reconfigurable Architectures; Particle Swarm Optimization; Modulo Scheduling; Loop-level parallelism; Mutation Operator; inner loop; Mapping.

Received Month Day Year; Revised Month Day Year; Accepted Month Day Year.

0. Introduction

Reconfigurable systems (Abielmona, 2005) have drawn increasing attention from both academic and commercial researchers in the past few years because they combine flexibility with efficiency and upgradability (Todman, Constantinides, Wilton, Mencer, Luk, & Cheung, 2005). Among reconfigurable architectures, many Coarse-Grained Reconfigurable Architectures (GGRAs) have been proposed as an alternative to FPGA-based systems (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003). CGRAs consist of programmable coarse-grained Functional Units (FUs) which support a predefined set of word-level operations; a programmable interconnection network; a configuration memory; and a controller (Vassiliadis & Soudris, 2007). Unfortunately the available parallelism has been exploited by few automated design and compilation tools (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003).

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 (Hatanaka & Bagherzadeh, 2007), 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. Iterations are started at a constant interval called the initiation interval (*II*). The time taken to complete a loop of n iterations is roughly proportional to *II*, thus the main goal of modulo scheduling is to find a schedule with a low as *II* as possible.

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 near optimal solutions with fast convergence and low execution time for various combinatory and multidimensional optimization problems (Abdel-Kader, 2008). We have used hybrid PSO with mutation operator to decide the placement and scheduling decisions in CGRAs. The MCHPSO algorithm has been tested on the benchmarks taken from (Texas Instrument inc, 1995; Park S. W., 2005; VLSI design laboratory, 2002). The benchmarks are derived from applications written in the C programming language. A shorter version of this paper was published as (Gnanaolivu, Norvell, & Venkatesan, 2010). 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 1. Modulo scheduling and PSO related work are discussed in Section 2. Our proposed PSO-based modulo scheduling algorithm (MCHPSO) is explained in Section 3. The experiments conducted are discussed in Section 4. Section 5 present the conclusion and future work.

1. BACKGROUND

In this paper, we propose an algorithm for modulo scheduling of loops to be mapped onto CGRAs. At the same time as it schedules, the algorithm places−assigns operations to FU −and routes− finds paths through space and time for data.

Each source program is converted from an imperative program 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 TA is replicated for many time cycles 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).

1.1 MOTIVATIONAL EXAMPLE

Figure 1 illustrates the compilation flow with a motivational example. Consider the architecture configuration shown in Figure 1 (a), and the 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. The final embedding of DFG in the RRG is shown in Figure 1 (d). The II for this example is 2.

The schedule produced by the algorithm maps each operation to a functional unit and a time and maps each edge in the DFG to a path in the RRG. Thus we are essentially searching for a graph homeomorphism (LaPaugh & Rivest, 1978), however with a couple of wrinkles. First, a new iteration starts every II cycles, any node or edge used in cycles i normally must not be used for another purpose in any cycle j such that $i \equiv j \text{ (modulo } II)$. Second, some edges or nodes in the TA may have capacities that exceed1. For example a register file may hold more than one word of data and it may be capable of multiple reads or writes in one cycle. Thus the real constraint is that for each TA node or edge r and for each t from0 $to\ II - 1$, the number of DFG nodes or edges mapped to the r in schedule cycles with remainder $t\ modulo\ II$ must not exceed $r's$ capacity. This constraint is checked with the aid of a modulo reservation table (MRT). The

columns of the MRT represent the resources of the TA and the rows represent remainder modulo *II*. Table 1 shows the completed MRT for the example of Figure 1.

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 WP1, RF1, and RP1, which are also reserved in the MRT. The capacity of resources are given in brackets, otherwise they have capacity of one.

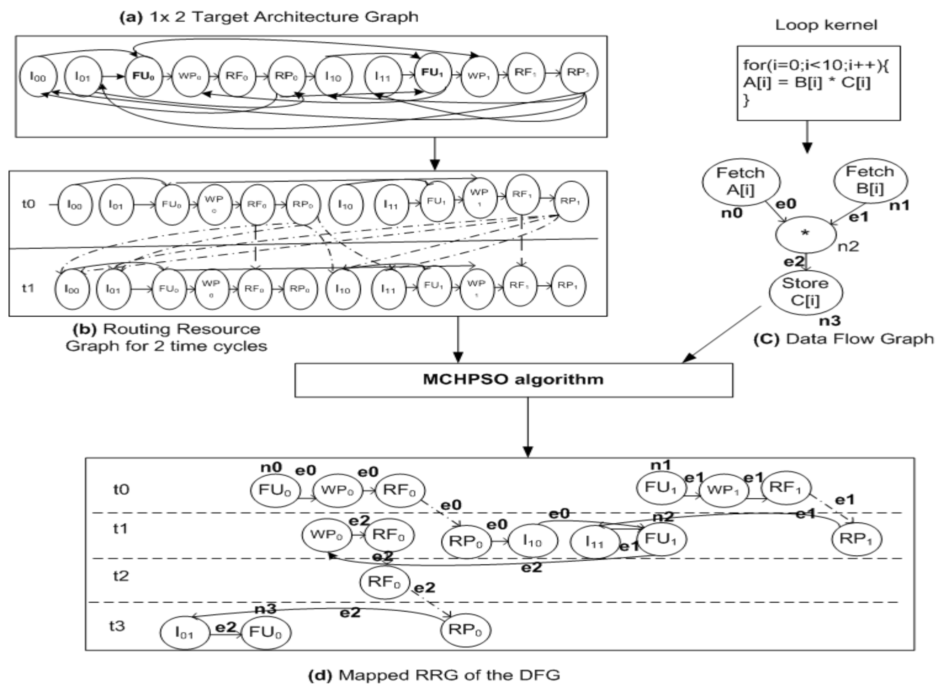

Figure 1. Motivating example a) 2 x 2 target architecture template instance, b) RRG, c) DFG and d) Final schedule, place and route result.

Table 1. MRT for the DFG in Figure 1 (c)

II \ resource	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

1.2 PARTICLE SWARM OPTIMIZATION

Particle Swarm Optimization (PSO) (Kennedy & Eberhart, 1995) 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 (Hu, 2006).

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

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

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

where

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

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 iteration i. c_1 , and c_2 denote the acceleration constants in the range $[0, 1]$, 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 iteration i , and 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. The PSO algorithm has the advantages of high speed, stable convergence and robustness; it parallelizes well and generates good solutions (Abdel-Kader, 2008).

When PSO is compared with Ant Colony Optimization (ACO) (Nonsiri & Supratid, 2008), PSO shows significant performance in the initial iterations and has the capability to quickly arrive at an optimal/near-optimal solution. An advantage of PSO over Genetic Algorithm (GA)

(Chatterjee & Siarry, 2006) is that PSO maintains all the solutions in the search space and requires less computational effort to arrive at high quality solutions. Since previous research (Abdel-Kader, 2008), (Chiang, Chang, & Huang, 2006) on PSO 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

1.3 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 (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003) and (Dimitroulakos, Galanis, & Goutis, 2007), our work aims to target a wide range of CGRAs. The ADRES (Mei, Lambrechts, Verkest, Mignolet, & Lauwereins, 2005) 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 (Vassiliadis & Soudris, 2007).

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

- V is the set of vertices. Each vertex represents a FU, RF, CB, RB 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 (Vassiliadis & Soudris, 2007). 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.

1.4 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$\in\mathbb{N}$ specifying the 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.
- X edges – Every edge e in the TA graph that doesn't end at a register write port is replicated across time.
- Y edges – Every 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 (Tuhin & Norvell, 2008).

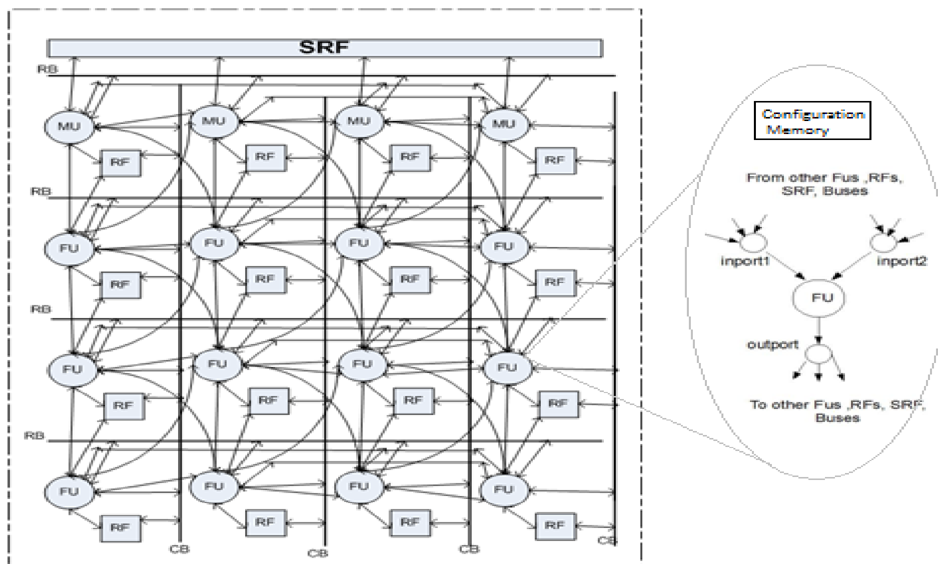

Figure 2. 4 x 4 target architecture template instance.

- Z edges – 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 cycle. Use of such an edge represents maintaining data in a register (Tuhin & Norvell, 2008).

1.5 MODULO SCHEDULING

Modulo Scheduling is a technique for software pipelining loops (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003). 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.

1.6 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 (Vassiliadis & Soudris, 2007). This interval is termed the initiation interval (*II*), essentially reflects 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 (Vassiliadis & Soudris, 2007).

1.7 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 (Tuhin & Norvell, 2008).

2. RELATED WORK

Software pipelining (Allan, Jones, Lee, & Allan, 1995; Lam, 1988) is used for instruction parallelism. The idea of software pipelining is to overlap execution of several iterations from the same loop. Iterations are started at regular intervals of *II*. After the first few iteration (the prologue) a repeating pattern of execution (the kernel) is repeated until the final iteration starts. Finally the pipeline is drained (the epilogue). List scheduling (Beaty, 1994) is a common approach used to solve the scheduling problem. Some of the approaches carried out in modulo scheduling of the loop body are discussed below.

The compilation of inner loop bodies for CGRAs has been done with DRESC (Dynamically Reconfigurable Embedded System Compiler) (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003), a retargetable compiler that is able to parse, analyze, place, route, and schedule C source code. In this work they propose a modulo scheduling algorithm based on simulated annealing (Wang, Wu, & Liu, 2001). This method can take a long compilation time for larger loops.

A memory-conscious mapping methodology for CGRA architectures was presented in (Dimitroulakos, Galanis, & Goutis, 2007) with data reuse capabilities and priority-based list scheduling algorithm. The resource aware mapping with local RAMs and flexible interconnection network enables the compiler to map the application. Most iterative modulo scheduling methods (Llosa, González, Ayguadé, & Valero, 1996; Rau, 1994) compute and analyze the dependence graph and orders the nodes to be scheduled. The idea of modulo scheduling is applied with a graph embedding (Heath, 1997) (Newsome & Song, 2003) technique using an affinity graph heuristic and skewed scheduling space in (Park, Fan, Kudlur, & Mahlke, 2006). This technique achieves better convergence and faster compilation times with dedicated register files and sparse network connectivity.

Some works have shown interest on different aspects in the loop body which are discussed below. Recurrence aware scheduling (Oh, Egger, Park, & Mahlke, 2009), which considers the dependence on the same operation while scheduling. Register constrained scheduling (Zalamea, Llosa, Ayguadé, & Valero, 2004) algorithm discuss the allocation of registers by using different register file models (Zalamea, J.; Llosa, J.; Ayguadé, E.; Valero, M, 2001). The register requirements are minimized by a heuristic strategy of hypernode reduction (Llosa, Valero, Ayguadé, & González, 1995) to shorten loop variant lifetimes without sacrificing performance.

The discrete problem of instruction scheduling has been solved using Particle Swarm Optimization PSO with the traditional list scheduling algorithm (Abdel-Kader, 2008). Since there is not much work done to improve the modulo scheduling algorithm with evolutionary algorithms, we started to try with simple PSO. The scheduling problem is NP-hard and needs a heuristic approach to find the solution. To avoid local optima from simple PSO and to represent the scheduling solution, we tried multi-dimensional PSO representation and its hybrid combination. To analyze the work done in (Gnanaolivu, Norvell, & Venkatesan, 2010), we used various TA topologies and FU configuration to validate hybrid PSO with mutation operator to decide the placement and scheduling decisions in CGRAs. 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 respecting modulo constraints and data dependence.

3. MAPPING ALGORITHM

3.1 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 as done in list scheduling (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003). In (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003; Tuhin & Norvell, 2008; Dimitroulakos, Galanis, & Goutis, 2007). 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 (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003) along with the combination of two heuristic approaches, PSO and randomization. From (Abdel-Kader, 2008) and (Chiang, Chang, & Huang, 2006) 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 (Uysal & Bulkan, 2008). To escape the local optima, we have used a randomization method in combination with PSO.

```
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
```

Figure 3. Mapping DFG to RRG

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. 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 for the given DFG. After generating the DFG and the RRG, the MCHPSO algorithm is executed to schedule, place, and route the loop.

3.2 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.

3.3 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 is given a random initial value for the place and time 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 (Abdel-Kader, 2008) 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 5.

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

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

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

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

3.4 FITNESS CALCULATION

The fitness calculation considers multiple objectives from the routing paths produced by Dijkstra's shortest-path algorithm (Dijkstra, 1959). 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 (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003), 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.

```
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

4. EXPERIMENT

4.1 SET UP

The 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 (Vassiliadis & Soudris, 2007), (ADRES) Architecture for Dynamically Reconfigurable Embedded Systems (Mei, Vernalde, Verkest, Man, & Lauwereins, 2003) was used for the experiments. The TA consists of two configurations, one is with a 4×4 grid as shown in Figure 2. The second configuration is with 64 FUs of 8×8 grid, which are divided into four tiles. Each tile consists of 16 FUs as same as in the 4×4 grid. The benchmarks used consist of ten programs, which are derived from (Texas Instrument inc, 1995; Park S. W., 2005; VLSI design laboratory, 2002).

4.2 EXPERIMENT RESULTS

The overall mapping results from MCHPSO of all the selected benchmarks on a 8×8 CGRA configuration is shown in Table 2 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

number 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).

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

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

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

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 of the loop. The largest loop kernels like IDCT_hor (horizontal pass) and FFT are scheduled within a maximum of 105.89 seconds.

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

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

From the mapping results, it is understood that the higher the number of loop operations, the larger the routing resources required. Our MCHPSO scheduling algorithm was able to map the Table 3 shows the overall mapping results from MCHPSO of all the selected benchmarks with a

4×4 CGRA. The first column shows the benchmark name, the second column denotes the number of operations in the loop kernel, and the third column shows the initiation interval (II) at which the loop body is mapped. The fourth column shows the schedule density without routing, as calculated in (8). The schedule density without routing considers the count of FUs used in the placement. The fifth column shows the schedule density of FU with routing, as calculated by (9). The routing path value for the fitness function is calculated from Dijikstra's algorithm to achieve better convergence and faster compilation times. The last column shows the time taken in seconds executed on a Intel Pentium M with 1 GB RAM and a clock speed of 1.73 GHz.

From the mapping results, it is understood that the higher the number of loop operations, the larger the routing resources required. Our MCHPSO scheduling algorithm was able to map the benchmarks both in 4×4 and 8×8 CGRA configurations. The *II* achieved to map the benchmarks were mostly minimum *II* or close to it.

4.3 ANALYZING THE USAGE OF FUNCTIONAL UNITS WITH DIFFERENT TOPOLOGY

The importance of having flexible interconnections in the FU are studied with various topologies as displayed in Figure 6. In Figure 6 (a) a mesh topology with four nearest neighboring FU connections is shown; Figure 6 (b) shows a meshplus1 topology, where it an enhanced mesh topology with additional interconnections between FUs of one hop; and Figure 6 (c) shows a star topology of eight neighboring FU connections. In a meshplus2 topology each FU connected to all FUs in the same row and column along with nearest neighboring connections.

Table 2. 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 3. MCHPSO -- overall mapping results for 4 × 4 CGRA

Bench-Marks	# of Ops	II	ScheduleDensity (without routing)	ScheduleDensity (with routing)	Exe Time in Seconds
FIR_cplx	25	3	50.00	68.75	0.84
latasynth	20	2	56.25	78.13	0.66
latanal	20	2	56.25	68.75	0.53
Volterra	28	4	43.75	57.81	1.36
IIR	36	4	56.25	78.13	2.17
IIR_biquad	35	5	43.75	61.25	1.77
8X8 IDCT_hor	78	7	68.75	89.29	7.20
4X4 FFT	67	7	56.25	81.25	9.86
8X8 FDCT_hor	74	7	68.75	90.18	6.45

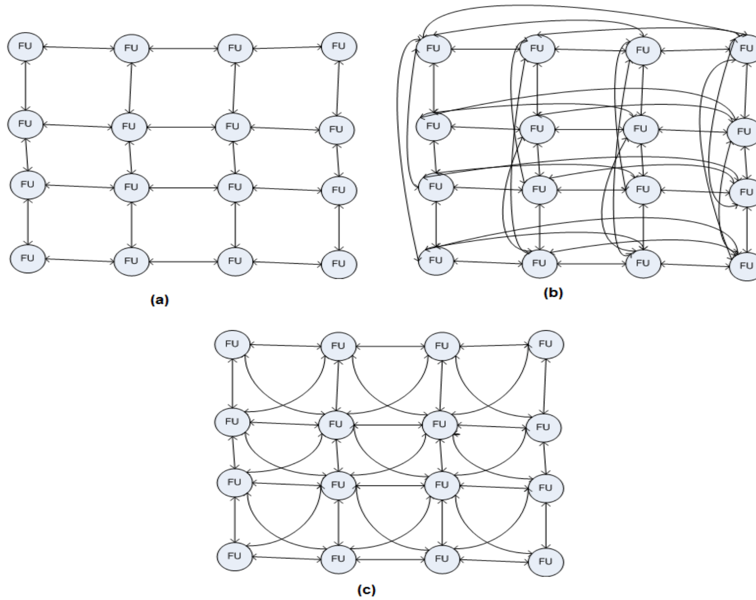

Figure 6. FU Topology are (a) Mesh topology (b) Meshplus1 topology (c) Star topology

Table 4 shows the comparison of using various topologies for the Functional Units. This experiment is done on a 4×4 CGRA. The first column shows the two benchmarks taken for comparison. We have chosen IDCT_hor and FFT benchmarks, because they are not able to schedule at minimum *II*. It would be a fair comparison of FU utilization in the previous initial intervals. The second column shows the minimum *II*. The third column shows the *II* tried and achieved to schedule without any overuse of resources. The fourth column shows the percentage of FU utilization at placement. The fifth, seventh, ninth, eleventh columns shows the percentage of FU overuse after scheduling, placement, and routing in mesh,meshplus1,meshplus2 and star topology. The sixth, eighth, tenth, twelfth columns shows the percentage of FU utilization after scheduling, placement, and routing in mesh, meshplus1, meshplus2 and star topology. From row1, row3 and row4 we can understand that the overuse of FUs is reduced when the interconnections are increased. The best utilization of FUs is achieved in the case of meshplus2 with star topology. When a benchmark has a lot of routing edges, the flexible interconnection

helps the MCHPSO scheduling algorithm to achieve a valid schedule with no overuse of resources.

Table 4. Utilization of Functional Unit interconnections with various topologies

				Mesh		MeshPlus1		MeshPlus2		MeshPlus2 and Star	
Bench-Marks	**Min II**	**II**	**P**	**O**	**P&R**	**O**	**P&R**	**O**	**P&R**	**O**	**P&R**
8X8 IDCT_hor	6	6	81.25	12.50	100.00	7.29	100.00	9.38	100.00	1.04	100.00
		7	*68.75*	*0.00*	*91.96*	*0.00*	*90.18*	*0.00*	*90.18*	*0.00*	*89.29*
4X4 FFT	5	5	81.25	28.75	100.00	28.75	100.00	22.50	100.00	20.00	100.00
		6	68.75	5.21	100.00	5.21	100.00	2.08	100.00	4.17	100.00
		7	*56.25*	*0.00*	*77.68*	*0.00*	*84.82*	*0.00*	*84.82*	*0.00*	*81.25*

4.4 ANALYZING THE USAGE OF REGISTER FILES WITH DIFFERENT INTERCONNECTIONS

The utilization of registers in the RFs is studied with different number of RFs and their interconnections as displayed in Figure 7. Each FU with its own dedicated RF is shown in Figure 7 (a); Figure 7 (b) shows the 4 RFs with each RF shared among four FUs; Figure 7 (c) shows each FU has a RF and the RF is shared among FUs adjacent in the diagonal directions.

Figure 8 shows the utilization of registers for the various register file topologies. This experiment is done on a 4×4 CGRA with each register file having four registers, four read and write ports. The shared 4 RFs topology uses the limited number of registers efficiently, but for large benchmarks such as the last two benchmarks, 8x8 FDCT_hor and 4x4 FFT, it overuses the registers nearly 20 to 100%. The shared 12 RFs topology utilizes the registers efficiently when compared with dedicated RF topology. Therefore, the shared 12 RFs topology works the best for all the benchmarks with no overuse of registers and efficiently uses the registers.

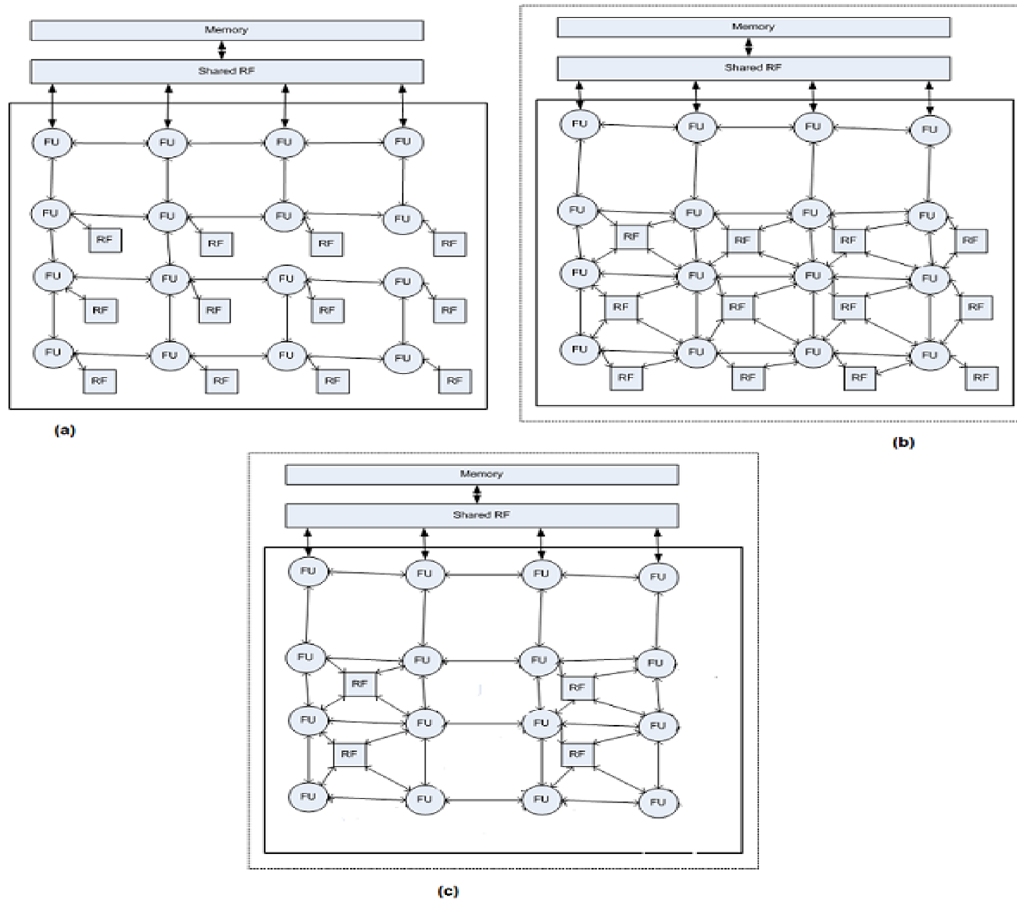

Figure 7. FU and RF Topology (a) 12 Dedicated RFs (b) 4 Shared RFs (c) 12 Shared RFs

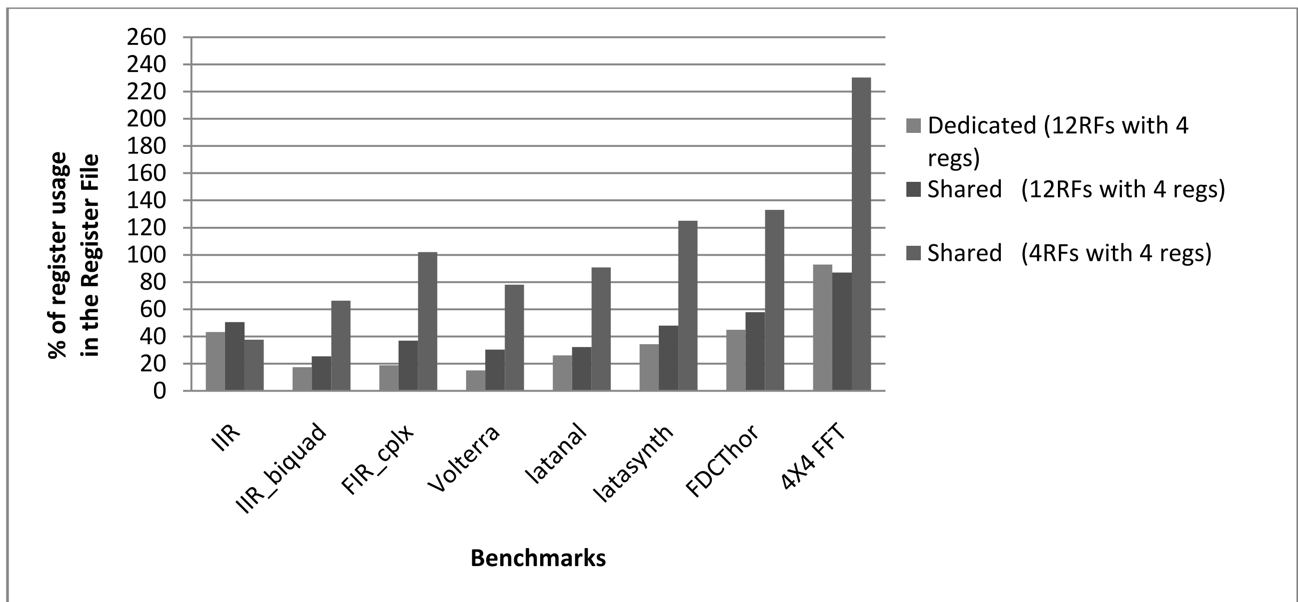

Figure 8. Percentage of register utilization in different topology

4.5 EFFECT OF VARYING PARTICLE SIZE IN MCHPSO ALGORITHM

To determine how many particles should be used in the MCHPSO scheduling algorithm, we conducted the experiments by varying the particle numbers in the algorithm. This experiment was done on an 8 x 8 CGRA in an Intel® Core™ i7-860 Processor with a clock speed of 2.8GHz utilizing all the four cores.

When we started with five particles, the algorithm wasn't able to come out of the local optimum of the best particle's fitness value. We then tried with ten particles and we were able to get the valid schedule.

To analyze the speedup of our MCHPSO scheduling algorithm, we compared the execution time of the algorithm utilizing two, four and eight processing threads on the quad core. Table 6 shows the speedup of MCHPSO algorithm on various benchmarks. The *first* column shows the benchmarks taken for comparison of using logical processors (P) in Intel i7 machine. The second to ninth columns show the execution time of MCHPSO algorithm. When using two processing threads each from different core, the speedup was more than 1.5 times that of a single processing thread. When using four processing threads each from four cores, the speedup was more than 2.5 times that of a single processing thread. When using all the eight processing threads, the speedup was more than 3.5 times that of a single processing thread execution. The multithreading available on the cores helped the algorithm to process the particle arrays faster. Our proposed MCHPSO works faster with more processing threads. The tail off of speedup is likely due to memory contention.

Table 5 shows the comparison of execution time with different particle sizes. The first column shows the three large benchmarks taken for comparison. The second to sixth columns show the

execution time for particle counts 10, 25, 30, 35 and 40. In all the particle count variations, we were able to get the valid schedule with same utilization of resources. Since we got the same utilization in all the different particle count, it seems that 10 particles are sufficient.

4.6 ANALYZING THE SPEEDUP OF MCHPSO ALGORITHM IN INTEL I7 PROCESSOR

The Intel Core i7-860 processor (Intel i7-860 processor, 2009) features four cores with a clock speed of 2.8 GHz. It features symmetric multithreading (hyper-threading) so that each core supports two threads, for a total of eight threads. It has maximum frequency of 3.46 GHz with Intel Turbo Boost technology

To analyze the speedup of our MCHPSO scheduling algorithm, we compared the execution time of the algorithm utilizing two, four and eight processing threads on the quad core. Table 6 shows the speedup of MCHPSO algorithm on various benchmarks. The first column shows the benchmarks taken for comparison of using logical processors (P) in Intel i7 machine. The second to ninth columns show the execution time of MCHPSO algorithm. When using two processing threads each from different core, the speedup was more than 1.5 times that of a single processing thread. When using four processing threads each from four cores, the speedup was more than 2.5 times that of a single processing thread. When using all the eight processing threads, the speedup was more than 3.5 times that of a single processing thread execution. The multithreading available on the cores helped the algorithm to process the particle arrays faster. Our proposed MCHPSO works faster with more processing threads. The tail off of speedup is likely due to memory contention.

Table 5. Variation of particle size on an 8 x 8 CGRA

Benchmarks	Execution time (in seconds) of MCHPSO with varying particle sizes				
	10	**25**	**30**	**35**	**40**
8X8 IDCT_hor	22.29	23.989	26.9	31.328	35.494
4X4 FFT	22.00	48.987	48.137	58.618	66.728
8X8 FDCT_ver	12.07	18.693	21.033	24.403	27.843

4.7 FUNCTIONAL UNITS CAPABLE OF ROUTING AND PERFORMING COMPUTATIONS

The computational resources in a CGRA are the functional units which are capable of executing a set of coarse-grained operations such as add, subtract, multiply, and shift. Initially we designed the FUs only to perform computation and to forward information during routing, if they are not performing any operation. Later we redesigned the FU to have additional ports and switches to perform computation and routing at the same time. We studied the utilization of FUs by comparing the two different FU configurations as shown in

Table 7. The first column shows the benchmarks taken for comparison. The second column shows the percentage of FU utilization with an FU configuration that cannot route when it is been used for execution. The third column shows the percentage of FU utilization with an FU configuration that can route and execute at the same time. The comparison shows that FU utilization decreases when they are capable of both routing and executing; this makes more resources available for mapping larger benchmarks.

Table 6. MCHPSO algorithm speed up comparison on a Intel i7 processor

Benchmarks	MCHPSO in Intel® Core™ i7 Processor Execution Time(Seconds)							
	One P	2 P's	3 P's	4 P's	5 P's	6 P's	7 P's	8 P's
FIR_cplx	7.29	4.23	3.10	2.98	2.79	2.68	2.20	2.08
latasynth	6.96	4.17	3.31	3.26	3.20	3.07	2.49	2.43
latanal	2.89	1.76	1.39	1.36	1.30	1.25	1.15	1.06
Volterra	6.26	3.45	2.59	2.36	2.34	2.17	1.86	1.76
IIR	9.13	5.37	3.92	3.65	3.54	3.32	2.81	2.68
IIR_biquad	13.60	7.61	5.40	5.12	5.16	4.56	3.98	3.68
8X8 IDCT_hor	79.31	42.44	32.24	28.82	28.33	27.51	22.69	22.29
4X4 FFT	84.46	44.23	33.16	31.54	29.65	27.58	22.73	22.00
8X8 FDCT_hor	23.28	13.14	9.97	9.39	8.77	8.41	7.15	6.94
8X8 FDCT_ver	44.28	23.97	18.23	17.12	15.87	15.02	12.30	12.07

Table 7. Comparison of FU utilization with placement and routing

Benchmarks	MCHPSO with FU that cannot route if used for execution	MCHPSO with FU that can both route and execute
FIR_cplx	42.19	39.06
Volterra	42.97	34.38
8X8 IDCT_hor	92.19	73.44
4X4 FFT	88.02	75.52
8X8 FDCT_hor	83.98	63.28
8X8 FDCT_ver	88.02	78.13

4.8 COMPARISON OF MCHPSO WITH OTHER MODULO SCHEDULING ALGORITHMS

Table 8 shows the comparative results of MCHPSO measured against the modulo scheduling algorithm for the ADRES architecture presented in (Vassiliadis & Soudris, 2007). 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 9 shows the comparison of MCHPSO with the modulo scheduling algorithm used in (Dimitroulakos, Galanis, & Goutis, 2007). 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. A memory-conscious mapping algorithm based on the priority-based list scheduling algorithm is used in (Dimitroulakos, Galanis, & Goutis, 2007). Therefore, we have compared the work done in (Dimitroulakos, Galanis, & Goutis, 2007) based on PEIT1 with our proposed algorithm. The first column in Table 9 shows the benchmarks taken for comparison. The second and fifth columns show the number of operations in the benchmark. The third and sixth column shows the *II* at which the algorithms

Table 8. Comparison of MCHPSO with results in (Vassiliadis & Soudris, 2007)

Comparing algorithms	**8 x 8 MCHPSO**					**Results reported in (Vassiliadis & Soudris, 2007)**				
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 9. Comparison of MCHPSO with results in (Dimitroulakos, Galanis, & Goutis, 2007)

Comparing algorithms	**4 X 4 MCHPSO**			**Results reported in (Dimitroulakos, Galanis, & Goutis, 2007)**		
Benchmarks	***# of Ops***	***II***	***Schedule Density (with routing)***	***# of Ops***	***II***	***Schedule Density (with routing)***
latasynth	20	2	78.13	18	6	75.00
Volterra	28	4	57.81	27	7	70.30
IIR	36	4	78.13	39	8	59.50
4X4 FFT	67	7	81.25	95	17	69.60
8X8 IDCT_hor	78	7	89.29	79	14	85.10
latanal	20	2	68.75	18	8	62.50

were able to map the benchmarks. 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 *II* for all benchmarks in spite of not using scratch pad memory, which has been used in (Dimitroulakos, Galanis, & Goutis, 2007). 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 (Dimitroulakos, Galanis, & Goutis, 2007). The numbers of operations

are different for the comparing algorithms because of the differing analysis and transformation phase carried out in (Vassiliadis & Soudris, 2007) and (Dimitroulakos, Galanis, & Goutis, 2007).

Our proposed algorithm finds schedules with a minimal *II* for all the benchmarks taken for comparison to the work done in (Vassiliadis & Soudris, 2007) with lower use of resources.

5. Conclusion and Future work

In this paper, we have done the analysis of the Modulo Constrained Hybrid Particle Swarm Optimization (MCHPSO) algorithm for the loop scheduling problem in CGRAs. The results from MCHPSO algorithm indicate that the algorithm can find a valid schedule, placement and routing for the given benchmark loops, often with a minimal initiation interval, and with a low use of resources. To study the parallelizability of the MCHPSO algorithm, we have executed it on a quad−core machine with eight logical processors and found good speedup. We also analyzed the MCHPSO algorithm with two different FU configurations. The experiment helped us to understand the enhancement in FU configuration increases the utilization of FUs. Various interconnections in all FUs study showed that increase in each additional edge produces a flexible routing process. Thereby, increases the utilization of resources. The size of RFs and its topology effect has been studied to know the usage of registers and which topology worked the best for our problem. Shared RFs with each FU gave the best utilization of registers. Since the MCHPSO algorithm depends on the particle solution, the number of particles to be considered is studied and reported.

The MCHPSO algorithm can also be enhanced to exploit conditional branches and inter-iteration dependence. The results produced by MCHPSO will be compared with other hybrid evolutionary algorithms in the future.

REFERENCES

(1995). Retrieved 2009, from Texas Instrument inc: http://dspvillage.ti.com/

Abdel-Kader, R. F. (2008). Particle Swarm Optimization for Constrained Instruction Scheduling. *VLSI Design , 2008*, 7.

Abielmona, R. (2005). *Reconfigurable Computing Architectures*. Retrieved 2009, from http://www.site.uottawa.ca/~rabielmo/personal/rc.html

Allan, V. H., Jones, R. B., Lee, R. M., & Allan, S. J. (1995). Software Pipelining. *ACM Computing Survey , 27* (3), 367-432.

Beaty, S. (1994). List scheduling: alone, with foresight, and with lookahead. *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*, (pp. 343-347).

Chatterjee, A., & Siarry, P. (2006). Nonlinear inertia weight variation for dynamic adaptation in particle swarm optimization. *Computers and Operations Research , 33* (3), 859-871.

Chiang, T., Chang, P., & Huang, Y. (2006). Multi-Processor Tasks with Resource and Timing Constraints Using Particle Swarm Optimization. *IJCSNS International Journal of Computer Science and Network Security , 6* (4).

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik , 1*, 269-271.

Dimitroulakos, G., Galanis, M. D., & Goutis, C. E. (2007). Design space exploration of an optimized compiler approach for a generic reconfigurable array architecture. *Journal of Supercomputing , 40* (2), 127-157.

Gnanaolivu, R., Norvell, T. S., & Venkatesan, R. (2010). Mapping loops onto Coarse-Grained Reconfigurable Architectures using Particle Swarm Optimization. *Soft Computing and Pattern Recognition, 2010. SoCPaR 2010. Proceedings., International Conference on*, (pp. 145-151). Dec 7-10, Paris.

Hatanaka, A., & Bagherzadeh, N. (2007). A Modulo Scheduling Algorithm for a Coarse- Grain Reconfigurable Array Template. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, (pp. 1-8).

Heath, L. S. (1997). Graph embeddings and simplicial maps. *Theory of Computing Systems , 30* (1), 51-65.

Hu, X. (2006). *PSO Tutorial*. Retrieved 2008, from http://www.swarmintelligence.org/tutorials.php

Intel i7-860 processor. (2009). Retrieved 2010, from http://download.intel.com/pressroom/kits/embedded/pdfs/Core_i7-860_Core_i5-750.pdf

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. *Neural Networks, 1995. Proceedings., IEEE International Conference on*, *4*, pp. 1942-1948 vol.4.

Lam, M. (1988). Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, (pp. 318-328). Atlanta,Georgia.

LaPaugh, A. S., & Rivest, R. L. (1978). The subgraph homeomorphism problem. *Proceedings of the tenth annual ACM symposium on Theory of computing* (pp. 40-50). San Diego, California, United States: ACM.

Llosa, J., González, A., Ayguadé, E., & Valero, M. (1996). Swing Modulo Scheduling: A Lifetime-Sensitive Approach. *In Conference on Parallel Architectures and Compilation Techniques (PACT'96)* (pp. 80-86). IEEE Computer Society Press.

Llosa, J., Valero, M., Ayguadé, E., & González, A. (1995). Hypernode reduction modulo scheduling. *Microarchitecture, 1995. Proceedings of the 28th Annual International Symposium on*, (pp. 350-360).

Mei, B., Lambrechts, A., Verkest, D., Mignolet, J.-Y., & Lauwereins, R. (2005). Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design and Test of Computers , 22* (2), 90-101.

Mei, B., Vernalde, S., Verkest, D., Man, H. D., & Lauwereins, R. (2003). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Computers and Digital Techniques, IEE Proceedings , 150* (5), 255-261.

Newsome, J., & Song, D. (2003). GEM: Graph EMbedding for Routing and Data-Centric Storage in Sensor Networks without Geographic Information. *Proceedings of the First ACM Conference on Embedded Network Sensor Systems* (pp. 76-88). ACM Press.

Nonsiri, S., & Supratid, S. (2008). Modifying Ant Colony Optimization. *Soft Computing in Industrial Applications, 2008. SMCia '08. IEEE Conference on*, (pp. 95-100).

Oh, T., Egger, B., Park, H., & Mahlke, S. (2009). Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not. , 44* (7), 21-30.

Park, H., Fan, K., Kudlur, M., & Mahlke, S. (2006). Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. *CASES ’06: Proceedings of the 2006 international conference on Compilers, architecture* (pp. 136-146). ACM Press.

Park, S. W. (2005). *Lattice LPC Analysis Filter*. Retrieved 2009, from http://www.engineer.tamuk.edu/SPark/Analysis-Synthesis.htm

Rau, B. R. (1994). Iterative modulo scheduling: an algorithm for software pipelining loops. *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture* (pp. 63-74). New York, NY, USA: ACM.

Todman, T., Constantinides, G., Wilton, S., Mencer, O., Luk, W., & Cheung, P. (2005). Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings - , 152* (2), 193-207.

Tuhin, M., & Norvell, T. S. (2008). Compiling parallel applications to Coarse-Grained Reconfigurable Architectures. *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference*, (pp. 1723-1728).

Uysal, O., & Bulkan, S. (2008). Comparison of Genetic Algorithm and Particle Swarm Optimization for Bicriteria Permutation Flowshop Scheduling Problem. *International Journal of Computational Intelligence Research , 4* (2), 159–175.

Vassiliadis, S., & Soudris, D. (2007). ADRES&DRESC: Architecture And Compiler For Coarse-Grain Reconfigurable Processors. In B. Mei, M. Berekovic, & J.-Y. Mignolet, *Fine- and Coarse-Grain Reconfigurable Computing* (pp. 255-297). Springer Netherlands.

VLSI design laboratory. (2002). Retrieved 2009, from http://www.vlsi.ee.upatras.gr

Wang, T. Y., Wu, K. B., & Liu, Y. W. (2001). A simulated annealing algorithm for facility layout problems under variable demand in Cellular Manufacturing Systems. *Computers in Industry , 46* (2), 181-188.

Zalamea, J., Llosa, J., Ayguadé, E., & Valero, M. (2004). Register constrained modulo scheduling. *IEEE Transactions on Parallel and Distributed Systems , 15* (5), 417-430.

Zalamea, J.; Llosa, J.; Ayguadé, E.; Valero, M. (2001). Modulo scheduling with integrated register spilling for clustered VLIW architectures. *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, (pp. 160-169).