COMPILING PARALLEL APPLICATIONS TO COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

Mohammed Ashraful Alam Tuhin

Department of Computer Science
Memorial University of Newfoundland
St. John's, NL A1B 3X5
maatuhin@ucalgary.ca

Theodore S. Norvell

Faculty of Electrical and Computer Engineering
Memorial University of Newfoundland
St. John's, NL A1B 3X5
theo@engr.mun.ca

ABSTRACT

In this paper a novel approach for compiling parallel applications to a target Coarse-Grained Reconfigurable Architecture (CGRA) is presented. We have given a formal definition of the compilation problem for the CGRA. The application will be written in HARPO/L, a parallel object oriented language suitable for hardware. HARPO/L is first compiled to a Data Flow Graph (DFG) representation. The remaining compilation steps are a combination of three tasks: scheduling, placement and routing. For compiling cyclic portions of the application, we have adapted a modulo scheduling algorithm: modulo scheduling with integrated register spilling. For scheduling, the nodes of the DFG are ordered using the hypernode reduction modulo scheduling (HRMS) method. The placement and routing is done using the neighborhood relations of the PEs.

Index Terms— Coarse-grained Reconfigurable Architecture, Modulo Scheduling, Routing Resource Graph, Graph Homeomorphism, Static Token

1. INTRODUCTION

Reconfigurable computing has been an active field of research for the past two decades. To overcome the disadvantages of Field Programmable Gate Arrays (FPGAs), many coarse-grained or ALU-based reconfigurable architectures have been proposed as an alternative between FPGA-based systems and fixed logic CPUs. Although CGRAs have the potential to exploit both hardware like efficiency and software like flexibility, the absence of proper compilation approaches is an obstacle to their widespread use. There has not been much work on compiling applications directly on to systems containing only CGRAs.

Compiling applications to CGRA, after the source code of the target application has been transformed and optimized to a suitable intermediate representation, is a combination of three tasks: scheduling, placement, and routing. Scheduling assigns time cycles to the operations for execution. Placement places these scheduled operation executions on specific processing elements. Routing finds routes to data from producer PE to consumer PE using the interconnect structure of the target architecture.

The main contribution of this paper is to formally define the compilation problem for CGRAs and propose a compilation approach for a family of CGRAs. The target architecture will be specified by the user. The intended application will be written in HARPO/L [1]. The input of the compilation is the intermediate representation of the target application in the form of a DFG and a description of the target architecture; the output will be executable code. HARPO/L is first compiled to a DFG representation [2]. The remaining compilation steps are: scheduling, placement and routing. The rest of the paper is organized as follows. Section 2 discusses how the input application will be presented to the back end of the compilation process. Section 3 describes our target architecture and an overview of our compilation process. Section 4 describes the modulo scheduling algorithm for cyclic portions of the target application. Section 5 describes the placement and routing steps performed during mapping from the input DFG to the input target architecture. Section 6 concludes the paper with possible future work.

Thanks to NSERC for funding.

2. EXECUTABLE DFG FROM SOURCE LANGUAGE

The input application is written using HARPO/L and then after some transformation and optimization the intermediate representation is obtained in the form of executable DFG.

2.1. Source Language Description

In [1] a language named HARPO/L (**HAR**dware **P**arallel **O**bjects **L**anguage) has been designed that targets CGRAs as well as microprocessors. HARPO/L is a structured language with a co construct used to express parallelism. It is a parallel, object-oriented, multi-threaded programming language. The language design allows explicit parallelism and enables the compiler to extract inherent parallelism.

2.2. Executable DFGs: The Input

We use the executable DFGs, represented by static token form for parallel program [2], as input for scheduling, placement and routing. We can represent our input executable DFG as a Graph $DFG = (N, E, op, inRole, outRole)$ such that:

- N is the set of nodes.
- E is set of edges labeled by $roles$. $e^{\leftarrow}$ represents the source of an edge, while $e^{\rightarrow}$ represents the target of that edge. *inRole* is a function $E \longrightarrow roles$, while $outRole$ is a function $E \longrightarrow roles$.
- *op* is a function: $N \longrightarrow operations$, which labels nodes with operations.

3. FRAMEWORK OF OUR TARGET ARCHITECTURE

We can represent the target architecture specified by the architecture designer by a graph $TA = (C, R)$, such that:

- $C = FU \bigcup RF$ is the set of functional units and registers.
- R is the set of interconnections where $r^{\leftarrow}, r^{\rightarrow} \in C$ for each $r \in R$.

For compilation purposes we have modeled our target architecture with routing resource graph (RRG). RRG is basically obtained by replicating the target architecture graph TA an infinite number of times and giving necessary interconnections across time cycles.

An RRG is a directed graph $RRG = (C \times \mathbb{N}, A \bigcup B \bigcup D)$. $A \bigcup B \bigcup D$ is the set of interconnections in a time cycle and across time cycles in a forward direction. Here $C \times \mathbb{N}$ is the vertices of the graph, i.e., resources of the TA replicated across time.

The set A, B, and D and the interconnection relations for their edges can be expressed as follows:

- $A = \{(i,r) | r^{\rightarrow} \in FU, i \in \mathbb{N}\}$
  $(i,r)^{\leftarrow} = (r^{\leftarrow}, i)$, for all $(i,r) \in A$
  $(i,r)^{\rightarrow} = (r^{\rightarrow}, i)$, for all $(i,r) \in A$
- $B = \{(i,r) | r^{\rightarrow} \in RF, i \in \mathbb{N}\}$
  $(i,r)^{\leftarrow} = (r^{\leftarrow}, i)$, for all $(i,r) \in B$
  $(i,r)^{\rightarrow} = (r^{\rightarrow}, i+1)$, for all $(i,r) \in B$
- $D = \{(i,f) | f \in RF, i \in \mathbb{N}\}$
  $(i,f)^{\leftarrow} = (f^{\leftarrow}, i)$, for all $(i,f) \in D$
  $(i,f)^{\rightarrow} = (f^{\rightarrow}, i+1)$, for all $(i,f) \in D$

4. FRAMEWORK OF OVERALL COMPILATION

Our target is to compile parallel applications to a given target architecture with optimal execution time. For that the target application is first written in HARPO/L. Then the source code is transformed and optimized to get the intermediate representation in the form of executable DFG. Here we have used the static token form for parallel programs. This executable DFG and the target architecture in the form of RRG is the two inputs of the back end of our compilation procedure. The output of compilation will be the executable code for the given CGRA and the given application.

Figure 1 shows the framework of our overall compilation.

4.1. Overview of our Compilation flow

After target architecture transformation and intermediate representation of the target application, we have two input graphs, an RRG and an executable DFG. Now our tasks is to map the DFG onto the RRG as efficiently as possible so that the execution clock cycle is as few as possible. We can consider the RRG as the source graph and the DFG as the target graph.

We will now give an overview of our compilation process. Our idea is to first analyze the DFG to extract some information that may be useful in the later phases. We are assuming here that the DFG given as input has been optimized using various common optimization techniques. The DFG is transformed by removing conditional branches and thus control dependences. For doing this we adopted the if-conversion method using predicates of [3] as is done in [4]. In If-conversion control dependences are converted to data dependences by computing a condition for executing each operation.

Since the input DFG can be cyclic, we need some approach for partitioning the cyclic and acyclic parts from the DFG. Then we will apply mapping (from DFG to RRG) for both the parts separately and integrate them for mapping as a whole.

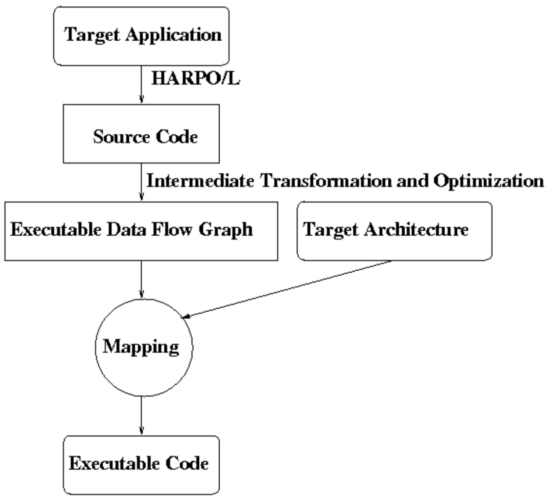

Fig. 1. Overall framework of our Compilation approach.

For mapping acyclic parts we will use the most commonly used list scheduling algorithm for resource constrained scheduling problems. In the list scheduling algorithm, instead of using the conventional priority functions, we will use the approach of [5].

Cyclic parts will be mapped using a register-constrained modulo scheduling method, improved modulo scheduling with integrated register spilling (MIRS) algorithm
[6]. For both the cyclic and acyclic parts, the nodes of the DFG will be placed to the processing elements of the target architecture and necessary routing is done accordingly.

4.2. Compilation Problem Formulation

Assuming all operation latencies are 1, we can formulate the scheduling, placement, and routing problem as one of finding a node disjoint subgraph homeomorphism (f_1, f_2) between the input executable $DFG = (N, E, op, inRole, outRole)$ and the RRG, modeled from the input target architecture, such that:

- $f_1(n) = (k, t)$ Here k is the resource (functional unit) which will execute n's operation. t is the execution time of n. $f_1(n)$ must be capable of executing n's operation, for all nodes $n \in N$. For any two nodes $u, v \in V$, $f_1(u) \neq f_1(v) = \phi$, if $u \neq v$.
- $f_2(e) = f_2(n_0, n_1) = P$, such that $start(P) = (k_0, t_0)$ and $end(P) = (k_1, t_1)$. Here $f_2(e)$ is a path from n_0 to n_1. $f_2(e)$ must be capable of carrying e's information, for all edges $e \in E$. For any two edges $e_0, e_1 \in E$, $f_2(e_0)$ is disjoint from $f_2(e_1)$ apart from endpoints, if $e_0 \neq e_1$.

But an operation can have latency greater than 1. So we need to generalize the above formulation. We can formulate our scheduling, placement, and routing problem as one of finding a pair of functions (f_1, f_2) between the input executable DFG and the RRG such that:

- $f_1(n) = \{(k,t), (k,t+1), ..., (k, t+\lambda_n - 1),\}$ Here k is the processing element which will execute n's operation. t is the start time when n will start executing and λ_n is the latency of n's operation. $f_1(n)$ must be capable of executing n's operation, for all nodes $n \in N$. For any two nodes $u, v \in V$, $f_1(u) \cap f_1(v) = \phi$, if $u \neq v$.

- $f_2(e) = f_2(n_0, n_1) = P$, such that $start(P) = (k_0, t_0 + \lambda_{n_0} - 1)$ and $end(P) = (k_1, t_1)$. $f_2(e)$ must be capable of carrying e's information, for all edges $e \in E$. For any two edges $e_0, e_1 \in E$, $f_2(e_0)$ is disjoint from $f_2(e_1)$ apart from endpoints, if $e_0 \neq e_1$.

Our formulated compilation problem has the following desired properties:

- n must be scheduled to be processed on a unique processing element $f(n) = c \in C \times \mathbb{N}$ starting at a unique time.
- A processing element k can process at most one node's operation at a given time t.
- If a node $n_1 \in N$ is a predecessor of another node $n_2 \in N$, then n_1 must complete its operation's execution before n_2's operation starts.

5. SCHEDULING

In this section we will discuss how to schedule, map and route cyclic parts, especially loops of an application. We will adapt Modulo Scheduling with Integrated Register Spilling (MIRS) [6]. MIRS is a software scheduling method that is capable of instruction scheduling with reduced register requirements, register allocation and register spilling in a single phase. But MIRS alone cannot do the required compilation for our problem. The reason is that MIRS does only scheduling and placement, it does not consider routing. We need to do routing during placement. The reason is as follows. During placement a cost function is computed to evaluate the quality of placement. While calculating that cost function, we need to incorporate routing cost. So we have modified the MIRS algorithm to incorporate this feature.

Another factor that we have incorporated into the MIRS algorithm is the consideration of loops with conditional branches. For doing this we have adapted the if-conversion and reverse-if-conversion idea from [3].

The input of the algorithm will be an executable DFG representing the cyclic part and the RRG. There are two outputs of the algorithm. One is the initiation interval (II). Another is a schedule of the nodes of the DFG, which is the pair of functions f_1 and f_2. This schedule will enable each node of the DFG to execute at its time cycle in its resource. Here f_1 is a partial function that maps each node of the DFG scheduled so far to a set of a pair of values: a time cycle and a resource and f_2 is a partial function that maps each edge of the DFG to a path in the RRG

5.1. Schedule_Place_Route

Figure 2 shows the actual scheduling, placement, and routing step of the IMIRS algorithm. In this phase a node is scheduled starting from a particular clock cycle in one or more resource(s) (functional unit(s) and/or register(s)). II and MII are initiation interval and minimum initiation interval respectively [7]. It first calculates the $Early\_Start_u$ and $Late\_Start_u$ of the node u to be scheduled, which produces a time frame in which that node can be scheduled legally. Suppose the $Start$ and End defines this time frame. For scheduling node u, Mapping() determines one or more resources in the RRG within this time frame starting from Start that produces optimal cost. During this checks are done so that there are valid routes from/to the predecessors/successors of u to u. Checks are also done so that there is no violation of dependence or no resource conflict. If such free resources are found, u is scheduled to the time cycles indicated from the position of the resource(s) in the RRG. Necessary updates are made to the partial schedule, resources, and registers. However, if no valid cycle is found, then the Force_and_Eject heuristic is applied. The partial schedule is scanned forwards or backwards depending on the values of $Early\_Start$, $Late\_Start$, II, and whether predecessors or successors of the node to be scheduled are already placed in the partial schedule. This is done according to the rules from [5]. The Force_And_Eject Heuristic, the Check_and_Insert_Spill Heuristic, and the Restart_Schedule Heuristic are given in detail in [6].

```
Procedure Schedule_Place_Route(DFG, RRG,
        $f_1$, $f_2$, $u$) {
    var $Start$, $End$;
    if ($Pred(u)$ is in Partial Schedule) {
        $Start = Early\_Start_u$;
        $End = Early\_Start_u + II - 1$; }
    else if ($Succ(u)$ is in Partial Schedule) {
        $Start = Late\_Start_u$;
        $End = Late\_Start_u - II + 1$;}
    else if (both $Pred(u)$ and $Succ(u)$ are in
        Partial Schedule) {
      $Start = Early\_Start_u$;
      $End = min(Late\_Start_u, Early\_Start_u + II - 1)$;}
    else {
        $Start = ASAP_u$;
        $End = ASAP_u + II - 1$;}
    if (not Mapping(DFG, RRG, $f_1$, $f_2$, $u$,
      $Start$, $End$))
        Force_And_Eject($i$, $u$); }
```

Fig. 2. Scheduling Phase of the IMIRS algorithm.

5.2. Improved MIRS for Compilation on CGRA

Figure 3 shows the pseudocode of improved MIRS algorithm for adapting to CGRA for cyclic parts. This algorithm uses the node ordering strategy of [5] for assigning priority to the nodes of the DFG. $Mapping()$, is used for placement of operations and routing them from producer FU to consumer FU in the available time cycles. The basic steps of the algorithm are summarized below.

At first the algorithm initializes the II with MII, f_1 and f_2 to empty functions. After the algorithm is completed f_1 will map all the nodes of the DFG with each node having one or more time cycles and a resource depending on the latency of the node's operations. $Budget$ is initialized to the number of nodes of the DFG times the $Budget\_Ratio$, where $Budget\_Ratio$ is the average number of times that each node of the DFG can be attempted to be scheduled with a fixed value of II.

After these initializations all the nodes of the DFG is ordered according to [5]. The ordered nodes are inserted into $Priority\_List$. Then the algorithm iteratively tries to schedule, place, and route operations from the $Priority\_List$. In each iteration, the operation with the highest priority is removed from the list and Sched-

```
Procedure IMIRS(DFG, RRG) {
    var $II$ := MII(DFG);
    var $f_1$ := empty();
    var $f_2$ := empty();
    var $Priority\_List$ := Order_HRMS(DFG);
    var $Budget := Budget\_Ratio \times$ Number_Nodes(DFG);
    while (!$Priority\_List$.empty()) {
        var $u := Priority\_List$.highest_Priority();
        $Priority\_List$.remove($u$);
        Schedule_Place_Route(DFG, RRG, $f_1$, $f_2$, $u$);
        if ($Priority\_List$.empty())
            Register_Allocation(DFG, $f_1$, $f_2$);
        Check_and_Insert_Spill(DFG,$f_1$,$f_2$,$Priority\_List$);
        if (Restart_Schedule(DFG, $Budget$))
            Re_Initialize($II++$,$f_1$,$f_2$,$Priority\_List$);
        else
            $Budget--$;
    }
    Generate_Code($f_1$, $f_2$, $II$);
}
```

Fig. 3. Improved MIRS algorithm for Compilation on CGRA.

ule_Place_Route() tries to find a FU for its execution using a route of free edges of the RRG that minimizes a cost heuristics. If such a FU and time cycle is found without violating any intra-iteration or inter-iteration dependency and resource constraints then those FU and time cycle are reserved for that operation so that they cannot be utilized by any subsequent operations until it is finished with utilizing them.. However, if no such cycle exists, then the algorithm employs the Force_And_Eject technique in which the node to be scheduled is forced to a specific cycle. Force_And_Eject, at the same time, ejects some nodes that were the reasons for dependency violations or resource conflicts.

Then the algorithm determines whether there is any need to spill values to memory to reduce the register pressure. The algorithm also detects the lifetime of a variable or its use which needs spilling. Then Restart_Schedule validates the current partial schedule with the current II. If the current partial schedule is valid then the algorithm continues with the next node of the $Priority\_List$, otherwise II is increased and the whole procedure is restarted with the new II.

After all the nodes of the $Priority\_List$ have been scheduled, the algorithm allocates registers for them. Then the configuration for executing the target application on the target CGRA is generated using the II and the mapping function f_1 and f_2.

6. PLACEMENT AND ROUTING

This section introduces our strategy for mapping from DFG to RRG used in the IMIRS algorithm. We have proposed a new placement method for CGRA. This method uses the neighborhood relations among the functional units (FUs) and registers. We will denote both FUs and registers as processing elements (PEs).

We can view the RRG as the given target architecture (composed of PEs) replicated across time. The interconnections among the PEs in a particular time and across time boundaries define regions with incrementing distances. All the unoccupied PEs in time cycle $Start$ can be viewed as the PEs of first choice. The reason for this highest priority is that those PEs can be reached from the producer/consumer PEs in the fewest possible clock cycle. Our idea is to look for a potential PE for a particular node from these PEs first, provided all the shortest route edges from producers or consumers of a node u to u are also unoccupied. All the possible PEs considering all the predecessors/successors of the node to be placed are tried and a cost function is evaluated for each of them. The PE with the lowest cost is selected for placing that node. If such a PE cannot be found, we will explore the PEs at time cycle ($Start$ + 1) and so on. That is, we choose the shortest paths connecting the producer PEs and the consumer PEs.

The algorithm is outlined in Figure 4. The first for loop deter-

```
Procedure Mapping(DFG,RRG,$f_1$,$f_2$,$u$,$Start$,$End$){
var $selected$, $j$;
var $time$ := -1;
var $Old\_Cost$ := Max_Num;
bool $found$ := false;
for $d$ := $Start$ to $End$ do {
    if $found$ break;
    var $Neighbors$ := Unoccupied PEs at time
            cycle $d$ such that for a PE all
            the edges along the shortest path
            from $PSP(u)$ or $PSS(u)$ to $u$
            in the RRG are unoccupied
    $found$ := !$Neighbors$.empty();
}
if not $found$ return false;
while (!$Neighbors$.empty()) do{
    $j$ := Select a neighbor from $Neighbors$();
    $Neighbors$.remove($j$);
    var $New\_Cost$ := 0;
    for each $v \in PSP(u)$ or $PSS(u)$ do
        $New\_Cost$+=Evaluate_Cost(DFG,RRG,$f_1$,$f_2$,$u$,
        $v$,$j$);
    if ($New\_cost$ < $Old\_Cost$) {
                $Old\_cost$ := $New\_Cost$;
                $time$ := $d$;
                $Selected$ := $j$;
    }
}
$f_1 := f_1 \cup \{u \mapsto \{(Selected, d)$, $(Selected, d+1)$,
                    ..., $(Selected, d+\lambda_u-1)\}\}$;
for each $v \in PSP(u)$ or $PSS(u)$ do{
    Let $P$ be the shortest path from the last
    (first) node in $f_1(v)$ to the first
    (last) node in $f_1(u)$
    $f_2 := f_2 \cup \{(v, u) \mapsto P\}$
}
return true;
}
```

Fig. 4. Algorithm for Mapping from DFG to RRG.

mines the possible candidates for mapping the current node. In each iteration unoccupied PEs at time cycle d, such that for a PE all the edges along the shortest path from PSP(u) or PSS(u) to u in the RRG are unoccupied, are elements in the neighbor set. Then the while loop selects the best PE from the candidates. Each neighbor candidate is considered for mapping and a cost function is evaluated for each of them considering all the elements in its PSP(u) or PSS(u). The PE that contributes the lowest cost is selected for placing the operation in question. Then the selected PE is marked occupied at

$Selected$ from time cycle d to $d + \lambda_u$-1. Necessary routing is done following the available interconnection that causes optimal routing. All the edges in RRG along the path P that corresponds to the edges between the selected PE at d and the PE occupied by each node in PSP(u) or PSS(u) is marked occupied.

6.1. Cost Evaluation

We use a greedy approach for evaluating the cost function of a particular placement. Our cost function consists of delay cost and interconnect cost. The delay cost of a node u is contributed by the time cycle in which the nodes $\in Pred(u)$ are scheduled. It is equal to the maximum of such delays. The interconnect cost comes from the interconnections that must be dedicated in order to route the node from the producer PE to the consumer PE. The longer the interconnections are occupied, the larger the $Early\_Start$ of the successor nodes will be. The PE with the lowest total of these costs will be selected for executing the current node.

7. CONCLUSION AND FUTURE WORK

In this paper a novel compilation approach for parallel applications to coarse-grained reconfigurable architectures has been proposed. The intended application is written in HARPO/L. The input of the compilation is the intermediate representation of the target application in the form of DFGs using static token and a description of the target architecture; the output is executable code. HARPO/L is first compiled to a DFG representation. The remaining compilation steps are a combination of three tasks: scheduling, placement and routing.

Some of the possible future works may be to implement the proposed compilation method for some benchmark parallel applications in the area of multimedia and embedded systems and to compare the compilation result with some of the related works. We also want to make the compiler retargetable across a wide range of target architectures.

8. REFERENCES

[1] Theodore S. Norvell, "HARPO/L: Language Design for CGRA project," www.engr.mun.ca/ theo, 2006.

[2] Dianyong Zhang, "An Intermediate Representation for CGRA Implementation," M.S. thesis, Memorial University of Newfoundland, 2007.

[3] J. C. H. Park and M. S. Schlansker, "On predicated execution," 1991, Hewlett Packard Laboratories, Technical Report HPL-91-58.

[4] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus, "Enhanced modulo scheduling for loops with conditional branches," in *MICRO 25: Proceedings of the 25th Annual International symposium on Microarchitecture*, Los Alamitos, CA, USA, 1992, pp. 170–179, IEEE Computer Society Press.

[5] Josep Llosa, Mateo Valero, Eduard Ayguadé, and Antonio González, "Hypernode reduction modulo scheduling," in *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, Los Alamitos, CA, USA, 1995, pp. 350–360, IEEE Computer Society Press.

[6] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "MIRS: Modulo scheduling with integrated register spilling," in *Proc. of 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*. August 2001, pp. 239–253, Springer-Verlag.

[7] B. Ramakrishna Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, New York, NY, USA, 1994, pp. 63–74, ACM Press.