A Hardware-assisted Modulo Scheduling, Placement and Routing Algorithm for Stream Computing in Coarse-Grained Reconfigurable Architectures

Waldir Denver Meireles Filho[*]
Departamento de Informatica
Universidade Federal de Vicosa
Vicosa, 36570-000, Brazil
waldir.filho@ufv.br

Ricardo Ferreira[†]
Departamento de Informatica
Universidade Federal de Vicosa
Vicosa, 36570-000, Brazil
ricardo@ufv.br

ABSTRACT

This work presents a hardware implementation of a on-the-fly algorithm which performs three tasks: scheduling, placement and routing for stream computing. The algorithm is a module schudeling heuristic for a coarse-grained reconfigurable architecture (CGRA). The algorithm maps a dataflow graph to compute data streams into a CGRA on the fly. The algorithm and the CGRA are implemented as a virtual layer over a commercial FPGA. The dataflow mapping is edge oriented. Each edge is mapped on average 6 clock cyles. The performance is two orders of magnitude better than a C-based FPGA soft-core implementation. Moreover, the proposed configuration unit has a very low cost, less than 1000 LUTs.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Reconfigurable Architectures, Run-time, Hardware-Assisted Algorithm

General Terms

Performance, Pipeline

Keywords

CGRA, FPGA, Placement, Routing, Scheduling, Modulo Scheduling

1. INTRODUCTION

Reconfigurable architectures are flexible as a software solution and has high performance as a hardware solution. There are two possibilities: FPGA and CGRA. A CGRA (Coarse-Grained Reconfigurable Architecture) consisting of a large number of word level functional units (FU). These architectures, in which the reconfiguration occurs at the functional level, ensuring greater flexibility while reducing the overhead of reconfiguration simplifying the mapping procedures. However, there are only few CGRA commercial architectures [7]. One solution is to implement a CGRA as a virtual layer over a commercial FPGA [4, 2]. Even for CGRA, the configuration memory could have a large size [1]. One approach is to generate the configuration at runtime and store only few configurations during the execution time. As the user application should be mapped to the target architecture on the fly, the mapping should be simple and direct. In addition, the mapping should be implemented at hardware level.

[*]Master Student
[†]Advisor

The goal is to develop an algorithm to map a dataflow in a reconfigurable architecture at runtime. The modulo scheduling algorithm or MSA [6] maps a dataflow by using pipeline and spatial computations. For instance, if the dataflow graph has 10 nodes and the architecture has only 5 units, the MSA could generate a pipeline implementation with at least two temporal partitions. Each partition will implement 5 nodes in the best case. The partition number will be the initial interval (II) to insert data streams at the pipeline datapath. Even if $II = 2$, the instruction level paralelism (ILP) will be high, $ILP = 5$ for the previous example. The MSA performs three steps: scheduling, placement and routing. As these problems are NP-complete, the MSA could be very time consuming as the approach proposed in [6]. Recently, a polynomial MSA heuristic by using a global interconnection network has be presented in [4]. This approach simplifies the placement and the routing complexity. The execution time is reduced to milliseconds and it could be implemented in a Just-in-Time (JIT) compiler.

We propose to simplify even more this previous work [4] by using a hardware assistant implementation of a modulo scheduling, placement and routing (MSPR). Instead of using a JIT compiler, our approach could be directly implemented at runtime. Two versions will be presented. The first one has only one temporal partition and the dataflow should be smaller to be fit into the CGRA. The second one uses more than one temporal partition when the dataflow is bigger than the CGRA. The experimental results show a fast and very low cost FPGA implementation of the MSPR algorithm.

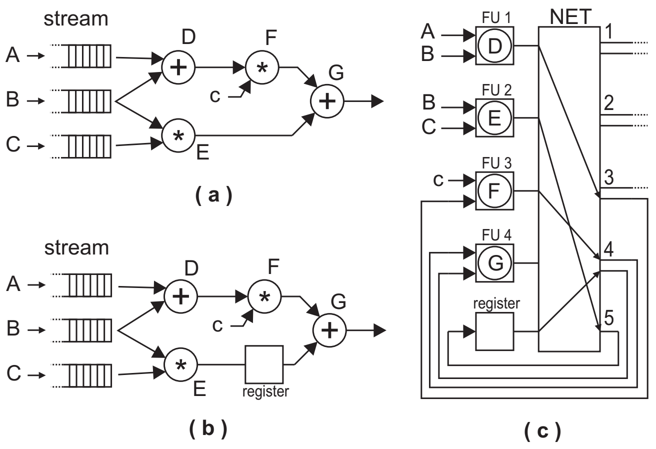

Figure 1: (a) DataFlow (b) Balanced (c) Mapping

2. SINGLE CONFIGURATION

The first algorithm does not handle any temporal partitions and it is an edge-oriented approach. Each dataflow edge represents a dependency between two operations. Each node is an operator such as an adder or multiplier which will be placed at a dedicated FU. The edges are implemented by network connection or the routing step. Each FU has one output and two inputs. The FU is connected to a global network.

The MSPR presented here is a greedy heuristic. This approach is based on the algorithm proposed in [4]. Each graph edge is visited only once, unlike previous work where during the scheduling, the nodes are visited more than once. Moreover, while the work of [4] is designed to be incorporated into compilers JIT (just-in-Time), our greedy heuristic is designed to be implemented in hardware. For more details about modulo scheduling and SPR algorithms, the readers are referred to [6, 5, 4].

The graph depicted in Fig. 1a shows the context of applications where data streams are received every clock cycle. The data are processed in pipeline fashion. The datapath should be balanced, which is achieved by inserting registers as shown in Fig. 1b. For this example, the data values from E would arrive early than the data from F to be computed in G. The nodes work in a synchronous manner and there is one register at each input (not shown).

The algorithm is implemented as an FSM and it uses only few registers and four small distributed memories: scheduling time, placement table, and two routing tables. At each step, a dataflow edge $x \rightarrow y$ is visited. The FSM flow is based on edge type. There are four scenarios (see Fig. 1a). The first scenario is an input edge as $A \rightarrow D$, then D is placed and scheduled at time 1. The second case is when D is already placed and the second edge should be routing as $B \rightarrow D$. In this case, only the stream B will be connected to D that has already been assigned to an FU. The third case is an internal edge as $D \rightarrow F$. First, F will be placed in a free FU. Then, routing connection $FU_D \rightarrow FU_F$ will be done. If F has already been visited and placed, only the routing will be done.

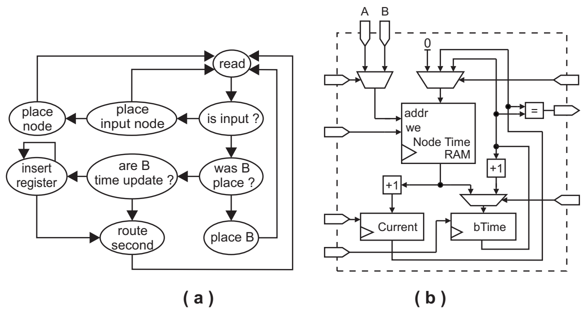

Figure 2: (a) FSM (b) Scheduling Memory

.

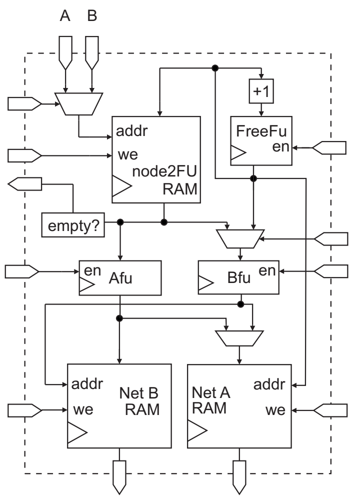

Figure 3: Place and Routing

The last scenario is the $F \rightarrow G$. As the edges are visited in breadth-first order, the edge $E \rightarrow G$ should have been visited prior to the edge $F \rightarrow G$. Therefore, a register, as shown in Fig. 1b should be inserted.

The FSM is illustrated in Fig. 2a. On average, three or four FSM states are traversed for each dataflow edge. The data structures for the scheduling, placement and routing are shown in Fig. 2b and Fig 3. As it is a hardware based implementation, the memories are distributed and many operations are performed in parallel. The memory NodeTime stores the node scheduling time. The memory Node2FU stores the placement table, where a node is mapped to a FU. Finally, two routing tables, one for each FU input, are used to the routing step (Net A and Net B in Fig 3). The other hardware resources are multiplexers, registers and simple operators (as an adder or comparator).

The whole system is illustrated in Fig. 4. This work focus on the MSPR implementation (black box) which processes on-the-fly a dataflow specification. The output of MSPR is sent to a reconfiguration memory which programs the CGRA to execute a stream application. Fig. 1c depicts the mapping of the first example in the CGRA. The CGRA is based on the proposed architecture presented in [4]. The CGRA consists on a set of FU and an interconnection network. For this work, we suppose a crossbar interconnection network. The

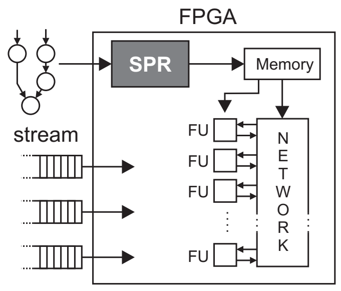

Figure 4: Reconfigurable System

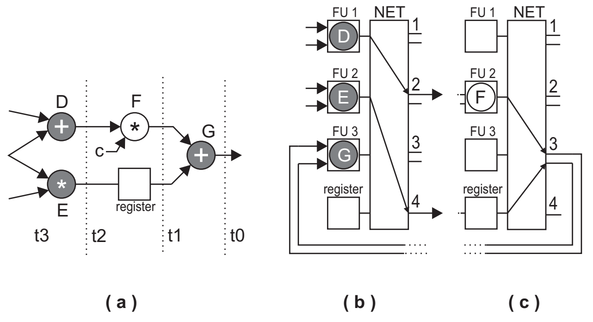

Figure 5: (a) Scheduling (b) Cfg 1 (c) Cfg 2

crossbar network simplifies the routing to a single assignment. However, the implementation cost is $O(n^2)$, which is a problem to implement large CGRA in FPGA, when the number of FU is bigger than 32 units. For large networks, a multistage as proposed in [4] could be used. An extra routing unit should be added too [4].

3. MULTIPLE CONFIGURATIONS

In previous section, we assume that the architecture has enough operators and registers to a single step dataflow mapping. Our greedy algorithm should be able to map a graph bigger than the target architecture. Let us suppose a CGRA with only three FUs. Let us consider the dataflow depicted in Fig. 1b, where there are five nodes including a register. Moreover each node has an output register which is not shown for ease of explanation. At least two temporal partitions are needed (see Fig. 5b and c). The nodes D, E and G are mapped in the first configuration (Cfg 1). The node F and the register are mapped in the second configuration (Cfg 2). The initial interval (II) to insert the data streams will be 2. The latency will be 3 and the throughtput will be 2 as the II. The ILP is at least 2. In addition, different FUs in the same configuration could process data in different time. For instance, node D computes a data at T_3 (the third element inside the stream), and node G placed in the same configuration is computing a data at time T_1 (first element) as shown in Fig. 5a.

This simple example has been used to explain the MSPR behavior when the size of dataflow is bigger than the CGRA. If the CGRA has 64 FUs, it is possible to map a 100 node dataflow in two temporal partitions as shown in next section. The II will be 2 but the ILP will be 50 and most FUs will be used.

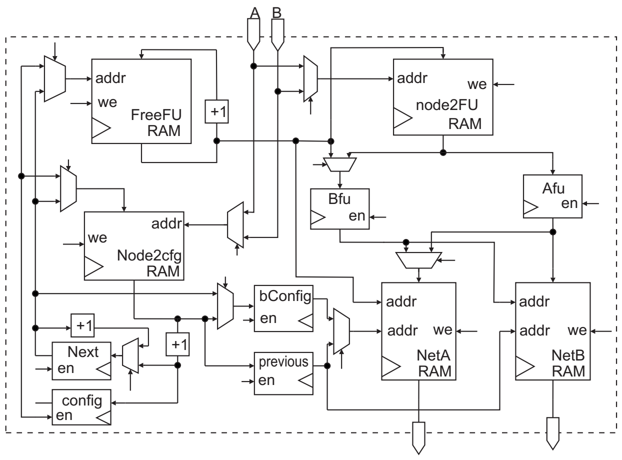

Figure 6: Hardware Resources for Temporal Partitions

The MSPR could handle more the one configuration by using few hardware resources. The FSM is quite the same as shown in Fig. 2a. The scheduling uses one more memory module to store the node temporal partition or configuration. The FU inputs are placed on the first partition P_0. When processing an edge as the $D \rightarrow F$ as FU_d belongs to P_0, a FU at P_1 will be allocated for F. Then, when processing partitions P_i the target operator will be placed at partition $P_{i+1} = (P_i + 1) \bmod P$, where P is the maximum number of partitions.

The routing need two memory modules for each temporal partition or a bi-dimensional memory. Fig 6 depicts the MSPR resources without the FSM module and the scheduling module. The memories Node2Fu and Node2Cfg store the placement and the configuration per node. FreeFU is used during the placement and a 2D memory is used to store the routing tables (Net A and B RAM).

4. EXPERIMENTAL RESULTS

The one step MSPR algorithm has been implemented in an FPGA Virtex6 as shown in Tab. 1. Column N shows the maximum number of nodes or FU supported for the target architecture. The source code has been written in VHDL. The code is parametrized as function of N. Columns LUT, FF depict the number of slice LUTs and slice registers after the P&R steps by using Xilinx Webpack ISE 14.2. For small values of N, the Xilinx tools automatically implements the memories by using LUTs. For large values of N, Column Mem shows the number of embedded RAM modules which are used to implement the MSPR. The FSM Column displays the maximum clock cycle. The MSPR uses very low resources as shown in Tab. 1, even for large values of N. For instance, MSPR uses only 199 LUTs for a target architecture with 64 FUs, which could compute up to 64 operations in parallel.

Tab. 2 depicts the resources for MSPR with several configurations. Column FU displays the maximum of CGRA Functional Units. As this MSPR version could fit graph bigger than the CGRA, Column *Cfg* depicts the maximum number of configurations or temporal partition supported by

Table 1: FPGA Resources for One Step

N	LUT	FF	Mem	Clk (Mhz)
16	82	59	0	302
64	199	123	0	275
256	519	315	4	262
1024	1973	1097	4	162

Table 2: 2,4,8 and 16 Configurations

N	FU	Cfg	LUT	FF	Mem
32	16	2	137	85	0
128	64	2	284	197	2
512	256	2	812	581	6
64	16	4	239	128	0
256	64	4	626	336	2
1024	256	4	2004	1096	11
128	16	8	368	203	1
512	64	8	949	594	3
256	16	16	723	334	3

the mapping procedures. Others Columns shows the FPGA resources similar to the Tab. 1. The size of MSPR depends on the maximum number of configurations or temporal partitions to be supported. Each aditional configuration will require two memory modules to store the routing tables. Even for several configurations, the MSPR uses few FPGA resources.

The MSPR algorithm has been tested over a set of dataflow graphs available in [3] as shown in Tab. 3. Columns *Name*, N show Benchmark name and node number. Column *Reg* shows the number of added register to balance the edges for pipeline execution. Column *FU* depicts the number of Functional Units of the target CGRA. Column MII shows the minimium pipeline initial interval. The minimum II is computed by $(N + Reg)/FU$. Column II shows the II reaches by the MSPR algorithm. If $MinII = 1$, the one step MSPR is used, otherwise more than one temporal partition will be computed. Column FSM displays the total number of clock cycles to execute the MSPR algorithm. Column MBlaze shows the number of cycles spent by a C equivalent implementation running on the MicroBlaze. The MicroBlaze is a soft processor core designed for Xilinx FPGAs.

The hardware MSPR version is at least 23× faster than the MicroBlaze C implementation for the smooth dataflow graph. For the best case, the arf benchmark, our hardware MSPR is up to 150× faster than a C based soft-core implementation. The FSM will requires only 172 clock cycles in comparison to the 25398 MicroBlaze cycles. Moreover, the MSPR uses less than 200 LUTs, while a MicroBlaze uses around 2000 LUTs. Therefore, the hardware version is up to three orders of magnitude faster than a soft-core C implementation. The MicroBlaze version used is MB 8.00.B. It is configured with barrel shifter and block memory of 128kb and without cache and floating-point units.

Let us consider collapse benchmark and the CGRA performance. As the $N = 65$, a pipeline version will execute in 65 clocks for each stream element. A parallel version, as the $II = 2$, will archive a throughput equals to 2. Therefore, the

Table 3: MSPR performance

name	N	Reg	FU	MII	II	FSM	MBlaze
arf	36	10	64	1	1	172	25398
collapse	65	38	64	2	2	374	22877
cosine2	112	62	64	3	3	591	46661
interpolate	156	40	64	3	4	1327	31376
smooth	260	128	256	2	2	1232	54963

ILP will be $N/2 = 32.5$, or 32 operations will be executed in parallel every clock cycle.

The CGRA reconfiguration time is at least 374 clock cycle (see Tab. 3). Then, if the input stream lenght is greater than $374/65 = 5.75$ or 6, the stream pipeline execution will be faster than the sequential pipeline. If the data stream is bigger, for instance a 1,000 element stream, the speedup will achieve a value close to 32×, for this example.

5. CONCLUSIONS

This work present a hardware-assistant implementation for a module scheduling, placement and routing algorithm. The algorithm has been implemented as an on-the-fly configuration unit for a CGRA. The experimental results show that the modulo scheduling unit could achieve a good performance (50 × faster than a softcore) by using few FPGA resources (less than 1000 LUTs). Future works will include less expensive interconnection networks as a multistage network proposed in [4].

6. REFERENCES

[1] T. Berticelli Lo, A. Beck, M. Rutzig, and L. Carro. A low-energy approach for context memory in reconfigurable systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE Int. Symposium on*, pages 1 –8, april 2010.

[2] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *IEEE/ACM CODES+ISSS*, 2010.

[3] ExPRESS. Electrical computer engineering dep., ucsb, usa. http://express.ece.ucsb.edu/benchmark/.

[4] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro. An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *International conference on Compilers, architectures and synthesis for embedded systems*, CASES '11. ACM, 2011.

[5] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. Spr: an architecture-adaptive cgra mapping tool. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 191–200, New York, NY, USA, 2009. ACM.

[6] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Conference on Design, Automation and Test in Europe*, page 10296, 2003.

[7] e. Volker Baumgarten. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *The Journal of Supercomputing (TJS)*, 26(2):167–184, 2003.