A Run-Time Modulo Scheduling by using a Binary Translation Mechanism

Ricardo Ferreira, Waldir Denver
Departamento Informatica
UFV
Vicosa, Brazil
ricardo@ufv.br

Monica Pereira
Departamento de Informatica e Matematica Aplicada
UFRN
Natal/RN, Brazil
monicapereira@dimap.ufrn.br

Jorge Quadros, Luigi Carro
Instituto de Informatica
UFRGS
Porto Alegre, Brazil
carro@inf.ufrgs.br

Stephan Wong
Computer Engineering Lab.
TU Delft
Delft, Netherlands
J.S.S.M.Wong@tudelft.nl

***Abstract*—It is well known that innermost loop optimizations have a big effect on the total execution time. Although CGRAs is widely used for this type of optimizations, their usage at run-time has been limited due to the overheads introduced by application analysis, code transformation, and reconfiguration. These steps are normally performed during compile time. In this work, we present the first dynamic translation technique for the modulo scheduling approach that can convert binary code on-the-fly to run on a CGRA. The proposed mechanism ensures software compatibility as it supports different source ISAs. As proof of concept of scaling, a change in the memory bandwidth has been evaluated (from one memory access per cycle to two memory accesses per cycle). Moreover, a comparison to the state-of-the-art static compiler-based approaches for inner loop accelerators has been done by using CGRA and VLIW as target architectures. Additionally, to measure area and performance, the proposed CGRA was prototyped on a FPGA. The area comparisons show that crossbar CGRA (with 16 processing elements) is 1.9x larger than the VLIW 4-issue and 1.3x smaller than a VLIW 8-issue softcore processor, respectively. In addition, it reaches an overall speedup factor of 2.17x and 2.0x in comparison to the 4 and 8-issue, respectively. Our results also demonstrate that the run-time algorithm can reach a near-optimal ILP rate, better than an off-line compiler approach for an n-issue VLIW processor.**

I. Introduction

The ever-increasing complexity of embedded system applications and the demand for combining many functionalities in a single system have increased the need for systems able to efficiently execute applications with heterogeneous behavior [1]. In order to efficiently execute these applications, it is necessary to find solutions able to identify (at run-time) the particular behavior of each application and use this information as a mechanism to improve performance. In this paper, we focus on run-time techniques and reconfigurable architectures to support inner loop processing. Moreover, the proposed run-time approach is based on binary translation mechanisms, and it could be extended to handle other application behaviors.

Nowadays, there is a large amount of streaming data mostly produced by sensors, telecommunication, and multimedia applications. These applications are implemented in general by using intensive loops. In addition, systems with different processing capabilities, ranging from embedded to exascale computing, require efficiency in terms of performance and power (Gops/W). Coarse-Grained Reconfigurable Architectures (CGRAs) have shown that they can provide both power efficiency and hardware acceleration [2].

In past years, many solutions emerged in an attempt to increase the loop performance by using Modulo Scheduling and CGRAs [2], [3], [4], [5], [6], [7], [8], [9], [10]. CGRAs are especially suitable for this, since they have a lower configuration overhead than fine-grained ones, such as FPGAs [11]. In spite of that, all solutions found in literature require special compilers or modifications in the application, which, in turn, precludes code reuse and software compatibility.

Recent works proposed the use of binary translation as a solution to reduce the intrinsic performance overhead of CGRA [12], [13]. Binary translation converts code compiled to a source ISA to run in a different ISA, in order to ensure software compatibility between different versions, or to allow application execution in different ISAs without the need for code recompilation. Additionally, run-time binary translation does not require compiler modifications, and may take advantage of optimizations that are not possible at compile time. Along with the possibility of optimizing the execution, run-time mechanisms are becoming essential due to the dynamic behavior of many applications, such as data-dependent computation, whose behavior may vary based on the inputs.

To fulfill the requirements of code reuse and software compatibility, we propose to apply binary translation (BT) onto the modulo scheduling (MS) approaches. To the best of our knowledge, no previous work has been carried out in order to define a BT run-time modulo scheduling algorithm for CGRAs. Moreover, a huge compile time reduction should be achieved, since this is the major challenge faced in previous modulo scheduling algorithms [2], [3], [4], [5], [6], [7], [8]. Recently, a low-complexity MS algorithm suitable for just-in-time (JIT) compilation was proposed in [9]. A CGRA with a crossbar network is used in [9] to reduce the complexity instead of mesh topologies [14], [2], [15], [4]. Nevertheless, the MS-JIT assumes that the starting point to perform the MS is a loop dataflow graph (DFG), and therefore it requires special JIT compilers or modifications in the application (like pragmas) to detect the loop and generate the DFG graph. In this work, we propose the first algorithm to detect, generate, and schedule the loop from the binary code. Another advantage of the proposed mechanism is its capability to benefit from the scaling process. For instance, if the memory bandwidth is improved, the binary translator could use this information to

978-1-4799-3770-7/14/$31.00 ©2014 IEEE

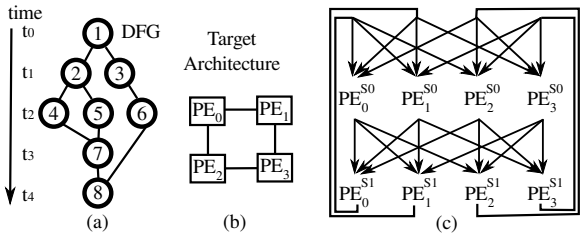

Fig. 1. (a) DFG (b) Target CGRA (c) Time Extended CGRA (TEC) graph

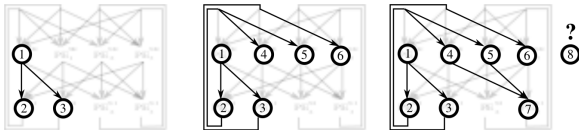

Fig. 2. (a) Partial TEC (b) Partial TEC (c) Final TEC

accelerate the application.

The remainder of this paper is organized as follows. Section II describes the modulo scheduling technique. Section III details the CGRA architecture. In Section IV, we present the proposed binary translation modulo scheduling. Experimental results are discussed in Section V. Section VI presents the works related to the proposed solution. Finally, Section VII presents conclusions and future works.

II. Modulo Scheduling

Modulo scheduling (MS) [16] is a software pipelining technique which overlaps different iterations of a loop to exploit a higher degree of Instruction-Level Parallelism (ILP). For ease of explanation, lets consider the data flow graph (DFG) and a 2x2 Mesh CGRA (depicted in Fig. 1(a-b)). The MS will schedule and map the DFG onto the target CGRA. Since this DFG has 8 nodes and the target CGRA has only 4 processing elements (PEs), the MS should use at least two temporal partitions. More formally, the DFG should be mapped onto the Time Extended CGRA (TEC) graph [2] depicted in Fig. 1(c), where there are two temporal partitions S_0 and S_1. This TEC represents all interconnections between the partitions for a 2x2 Mesh. Each CGRA connection $PE_a \rightarrow PE_b$ creates a TEC interconnection $PE_a^{Si} \rightarrow PE_b^{Sj}$ where $j = mod(i+1, P)$, P is the number of partitions, and mod is the modulo function. Moreover, if the PEs have internal registers, all PEs have self-connections between the partitions.

Considering the DFG where node 1 is connected to nodes 2 and 3. If node 1 is assigned to S_0, nodes 2 and 3 should be placed into S_1. Fig 2(a) depicts one possible partial scheduling and mapping. Since nodes 2 and 3 are placed in S_1, then their successors should be placed in S_0 by the MS algorithm as depicted in Fig 2(b). Although the TEC has 8 PEs, the MS fails, since node 7 is placed in S_1 and there is no free PE in S_0 to map its descendent node 8 (see Fig 2(c)).

The MS is similar to the set covering problem, where the node scheduled at time t_i is placed in the partition set $S_{mod(i,P)}$. For instance, the nodes at t_0, t_2, t_4 will be mapped onto the S_0 set. In this example, the MS fails since there are only 4 PEs for five nodes (1,4,5,6, and 8) in the S_1 set. Therefore, it is not possible to perform the MS with

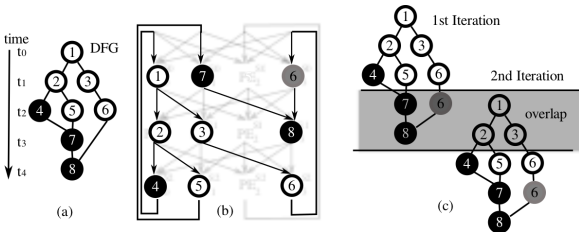

Fig. 3. (a) DFG (b) TEC graph (c) Loop Overlapping

the minimal number of partitions, and consequently, the DFG should be re-scheduled or local registers should be used. In order to find the minimal sets, the EPImap and REGImap algorithms [2], [10] proposed to use recomputation and local registers. For instance, PE_2 and PE_3 could store the results of nodes 6 and 7 in the local register file to forward these values to node 8 placed at $PE_2^{S_1}$. If it is not feasible with the minimal set, the partition number is increased until a solution is found. Only recently, the problem of mapping a DFG into a TEC CGRA has been proved to be NP-Complete in [2]. Therefore, due to the complexity of mapping the DFGs, all MS approaches are off-line and compiler-based, except the MS-JIT approach proposed in [9]. The MS-JIT applies two strategies to reduce the mapping time. Firstly, the target architecture is a cross-bar based CGRA to reduce the complexity of the placement and routing steps, from NP-complete to $O(1)$. However, the scheduling step itself is still NP-complete. Secondly, the MS is implemented by using a greedy approach based on graph traversal, where each node is visited once. Nevertheless, the MS-JIT approach does not include the DFG generation, and an off-line compiler is still needed.

Assuming the DFG example depicted in Fig. 1(a), MS has several constraints to take into account. First, the DFG paths should be balanced since the execution is performed in a pipelined fashion. Therefore, a buffer node should be inserted in the edge $6 \rightarrow 8$. In addition, in relation to memory operations, which are the most severe constraints, it is assumed that nodes $4, 7$, and 8 are memory operations and the CGRA can perform just one memory access per cycle. Therefore, at least three partitions are needed as depicted in Fig. 3. Two loop iterations will execute at the same time in a pipelined fashion. Every third clock cycle, a new iteration is started. The scheduling quality is measured by the minimal number of partitions, which corresponds to the throughput. In this example, the solution is optimal and it is bounded by memory constraints.

III. Architecture

In this work, the proposed architecture is based on the homogeneous CGRA presented in [9]. However, in this work, we propose a tightly accelerator approach and a heterogeneous CGRA as shown in Fig. 4(a). The CGRA copies the values from the CPU register file to the CGRA inputs, then the loop body is executed, and finally, the output values are written back to the register file. A monitor module detects the loop during the execution. The BT module is implemented in software and it translates and generates the CGRA configuration on-the-fly.

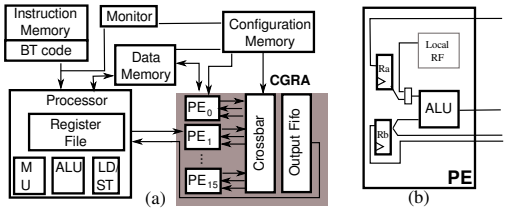

Fig. 4. (a) Target Architecture (b) Processing Element (PE)

The monitor could be implemented in hardware by using a FSM like a decode stage. The goal is to dynamically detect inner loops, assuming loops based on a backward goto. When a backward goto is executed, a monitor is switched on until the goto is reached again. The monitor works in parallel with the CPU. A loop is a candidate to be mapped if all instructions inside the loop are supported by the CGRA. In this work, all logic/arithmetic and load/store instructions are supported. Simple conditional assignments are also supported, as well as branch instructions to outside the loop body (exit points). No floating point instructions are supported. However, the monitor can be extended to detect more complex loop structures.

The target CGRA is a heterogeneous architecture interconnected by a crossbar network. A heterogeneous PE set reduces the area cost and configuration bits. In this work, we assume three PE types: multipliers, load/store, and ALU units. Moreover, the architecture has 16 PEs in total, since the throughput does not increase that much beyond the size of 16 as shown in [14]. In addition, the crossbar of 16 units is feasible even though its cost is $O(n^2)$.

Fig. 4(b) depicts the internal structure of an ALU unit. Each PE has two input registers R_a and R_b. These registers are also used as buffer registers (BR) as presented in [9], and in this case the ALU is bypassed and it can not be assigned to any DFG node. While REGImap [10] uses the local register file (LRF) as BR in parallel to the local ALU, in our approach, the LRF only stores the immediate operand and loop input values. The goal is to simplify the MS algorithm to be suitable at run-time.

IV. Binary Translation Modulo Scheduling

In [17], a BT mechanism was proposed to dynamically map MIPS code onto a CGRA. The proposed BT unit implements a dynamic scheduling algorithm for a CGRA similar to a Tomasulo algorithm used in superscalar out-of-order processors. All blocks could be mapped onto a large CGRA, and a configuration cache is used to store the most common blocks. In spite of that, a MS approach is more efficient than BT [17] and Tomasulo for loops. Considering an inner loop with 32 instructions mapped by the BT proposed in [17]. Suppose that the achieved latency is 8 cycles for this loop. The achieved ILP will be $32/8 = 4$. The CGRA should have at least 32 units, and these units are not used in pipelined fashion. On the other hand, the MS algorithm overlaps loop iterations. Considering a MS CGRA with 16 units. Suppose there is a feasible scheduling with two temporal partitions. Every two cycles, a new iteration is processed. In this case, the ILP will be $32/2 = 16$. Thus, the CGRA size is reduced by a factor of $2\times$ (16 units instead of 32 units), and the performance is improved $4\times$ by the MS approach in comparison to BT [17] for inner loop acceleration.

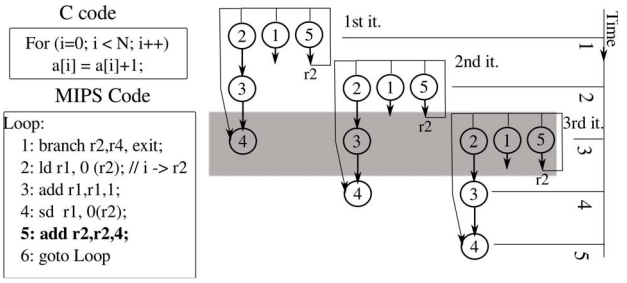

Fig. 5. Simple Vector Increment

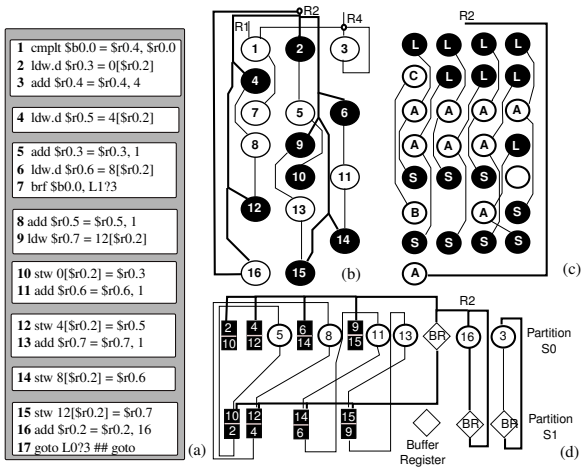

Fig. 6. (a) VLIW 4 code (b) One L/S Compiled DFG (c) 4 L/S Compiled DFG (d) 4 L/S 2 partitions CGRA

Although the MS is more suitable for loops, the MS binary translation (MS-BT) is more complex. In addition to detect the RAW, WAR, and WAW hazards, the MS-BT should detect the recurrence values between the iterations, taking into account the temporal partitions and balancing the DFG paths by inserting buffer registers. Moreover, if the schedule fails, the MS-BT should increase the partition number. Therefore, our MS-BT differs from previous Tomasulo and the BT approach proposed in [17] to dynamically solve new challenges. Moreover, as all previous MS approaches are compiler-based, this is the first compiler-free binary translation approach.

Before describing the MS-BT approach, first, we present a simple loop example. Initially, this example is compiled considering only one memory access operation per clock cycle. Then, we compare our MS-BT dynamic approach to a VLIW compiler-based approach. The capability to perform more memory operations per clock is modified to show adaptability of our approach in comparison to static compiler-based approaches. In the example, we assume a simple loop to increment the values of one vector. The C code and a pseudo MIPS code are depicted in Fig. 5. Fig. 5 also presents the correspondent dependence graph for the ideal case of a software pipelining execution with overlapped iterations. For ease of explanation, the instructions are numbered. The goto instruction is not depicted. The recurrence dependence on R2 value is shown by the feedback edge. A loop iteration can be executed in one clock cycle if the hardware supports the execution of all operations at same time (as shown at time 3).

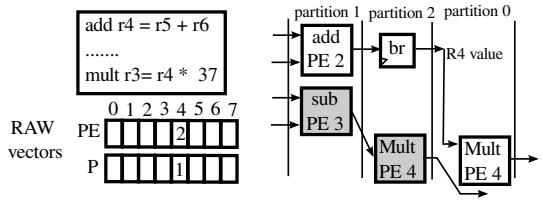

Fig. 7. RAW example with Buffer register

It is well known that loop unrolling and VLIW processor reaches high ILP for inner loops. Fig. 6(a) depicts the assembly code generated by a VLIW compiler [18] for a 4-issue VLIW processor. The compiler uses an unrolling factor of 4. The code has 8 VLIW instructions. Each instruction can have up to 4 operations (grouped inside each white block). Each vector element should be read (load) and written (store), therefore at least 8 instructions are needed as only one memory access operation per clock cycle is allowed. In this example, the compiler reaches the optimal ILP, and one element of the vector is added every two clock cycles. Fig. 6(b) depicts the DFG. There are 4 dependence load-add-store chains, register R_2 is the memory index and R_4 is the loop counter. Now, let us assume that, thanks to some technology improvement, the architecture can perform 4 memory access operations (ops) per clock cycle. To include this information into the VLIW system, one is forced to recompile the VLIW code. Fig. 6(c) depicts the generated code for 4-issue and 4 memory ops per clock cycle. The code has 8 VLIW instructions and unrolling factor of 8. There are 8 load/add/store chains ($L \rightarrow A \rightarrow S$). Therefore, eight elements are added per iteration or one element per clock cycle, which doubles the ILP. However, if the architecture could perform 4 memory ops per clock cycle, it is possible to produce one loop result per $\frac{1}{2}$ cycle. Therefore, the compiled solution is 50% of the optimal solution.

Our proposed MS-BT mechanism adapts the dependence graph as a function of the available resources. For this example, there are 8 memory access operations to be executed and the architecture only supports 4 per clock cycle. Therefore, at least 2 partitions are needed. Assuming that the load/store (L/S) are executed in a two-stage pipeline unit. The first stage computes the address and the second stage send/receives data to/from the memory. Despite a latency of 5 cycles to compute a load-add-store chain, the throughput found by our approach is 2 clock cycles due to the iteration overlapping, as depicted in Fig. 6(d). Moreover, as 4 elements are processed in parallel, the loop throughput is 4 elements/2 cycles, or 1 element per 1/2 cycle, which is the optimal solution.

The binary translation algorithm scans the instruction in order. In addition to RAW detection, the MS-BT performs buffer register insertion and cycle recurrence register detection. A RAW is verified by using a target unit vector indexed by the register number. Supposing that the current RAW instruction is a multiplication, if there is no multiplier unit in the current partition, buffer registers (BR) will be inserted until finding a free multiplier unit. Fig. 7 depicts an example of RAW in R4 between an add instruction in partition 1 and a multiplication instruction. The R4 will be computed by PE_2 in partition 1. Assuming only one multiplier unit (PE_4) per partition, since the multiplier is already allocated at partition 2, one buffer register will be inserted, and the multiplier will be allocated in next partition as depicted in Fig. 7.

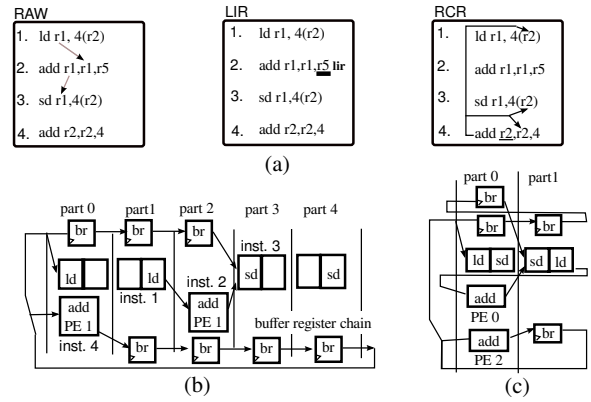

Fig. 8. LIR and RCR registers

In addition to the RAW (read after write) dependence, the MS-BT should be able to detect the loop input registers (LIR). A LIR is a register that is read at least once and it is not overwritten, and it represents invariant loop input values. Finally, there is the recurrence cycle register (RCR), which is similar to a LIR, however it is overwritten. The RCR can be a loop counter, vector index, or inter-iteration values. Fig. 8 depicts an example of LIR and RCR inside a simple loop, which performs the load-add-store chain and increments the vector index. Register R_5 is a LIR, and it behaves as a constant during the loop execution. Register R_2 is an RCR. However, since the instructions are processed in order, R2 behaves as a LIR until the MS-BT processes the last instruction. By default, all registers that appear as source registers are considered as LIR, until they are overwritten and become a RCR.

Moreover, there is register R_1 in Fig. 8, which is a false output register, since it only carries temporary values due to RAW dependences. For each LIR, a list of dependence functional units is created during the loop scanner. An RCR also has a dependence list.

For ease of explanation, let us assume that the scheduling is done by using 5 partitions (0 to 4), as depicted in Fig. 8(b). The two-stage load is executed at PE_0 in partitions 0 and 1. The result is sent to the adder at PE_1 in partition 2, and finally, it is sent to the two-stage store at PE_0 in partitions 3 and 4. The adder at PE_1 is used in partition 0 to execute instruction 4 (add R2) and in partition 2 to execute instruction 2 (add R1), as the units are time-multiplexed. The LIR list will also generate the buffer register chains for the RCR.

Register R_2 will generate a chain of three buffer registers (BRs), as the R_2 value is used in partition 0, in PE_0 (load), and also in PE_0 (store) in partition 3. Moreover, an additional 4 BR chain is generated to send back the value to the adder. Although Fig. 8(b) depicts 7 BRs in total, the target architecture uses only 2 BRs, since the BRs are time-multiplexed, and the maximum number of BRs is the maximum number per partition. For this example, partitions 1 and 2 need only two BRs.

For this example, a better scheduling is possible. Assuming only one memory access operation per clock cycle, the minimal

```
Inst = Fetch Instruction()
While ( Inst != END )
  type = get InstructionType(inst), Partition = initial
  if (Immediate Operand)
    if (Rs1 == input)
      PE = get_FreePE(type,Partition)
      Place  PE[partition] = Inst
      LIR.insert(Rs1,partition,PE)  // added LIR list
    else
      PEraw = get PE[Rs1]      // Read-after-Write
      p = partititon(PEraw)
      PE = get_FreePE(type,p)
      Place  PE[p] = Inst,          Route PERaw -> PE
  else Two Value Operands:
    if (Rs1 == input && Rs2 == input)
      PE = get_FreePE(type,Partition), Place  PE[partition] = Inst
      LIR.insert(Rs1,partition,PE)  // added LIR list
      LIR.insert(Rs2,partition,PE)  // added LIR list
    else    // Read-after-Write
      PEraw = Later PE(rs1,rs2),
      p = partititon(PEraw)
      PE = get_FreePE(type,p),
      Place  PE[p] = Inst
      if (Rs1 == input ) LIR.insert(Rs1,p,PE)
      if (Rs2 == input ) LIR.insert(Rs2,p,PE)
      Route PERaw ->  PE,
      Verify RCR(Target Register)
  Inst = Fetch Instruction()
end While
Place and Route   LIR lists
```

Fig. 9. Binary Translation Algorithm - Pseudo Code

number of partitions is 2. Fig. 8(c) depicts the mapping by using two partitions. Similar to the example depicted in Fig. 6(d), the load is mapped in partitions 0 and 1, then the adder in partition 0, and finally the store in partitions 1 and 0, respectively. Every two clock cycles, the loop produces a new value. At resource level, the usage is maximum in partition 0, where the PE_0 (load/store), PE_1 and PE_2 (ALUs), and two BRs are needed. It is important to notice that the R2 LIR chain has also three BRs as the previous mapping with 5 partitions depicted in Fig 9(b). However, there is an overlapped iteration, and the BRs in partition 0 store values of different iterations.

Fig. 9 depicts a pseudo-code of our modulo scheduling - binary translation algorithm. The instructions are scanned in order. There are two basic instruction types: one operand (plus an immediate operand) and two operands. In this algorithm, we assume the notation Rs1 and Rs2 for the source register operands. As well as in Tomasulo algorithm, a Read-After-Write (RAW) is verified by using a target unit vector indexed by the register number. When a RAW is detected, the current instruction receives a forwarding value. Supposing that the current instruction is a multiplication, if there is no multiplier unit in the current configuration, buffer registers (BR) will be inserted until finding a free multiplier unit. Then, the instruction is placed, and in case of RAW or BR registers, the connections are routed between the temporal partitions. If the instruction has an input register R, the $PE[p]$ is inserted in LIR list of R, where PE is the processing element where the instruction is placed, and p is the temporal partition. Next, the instruction target register is verified to detect RCR registers, and the next instruction is fetched until all instructions are processed. Finally, all LIR registers are generated, placed and routed. Similar to Tomasulo algorithm, the register number is replaced by the unit number to eliminate the output and anti-dependences, i.e. WAW and WAR hazards.

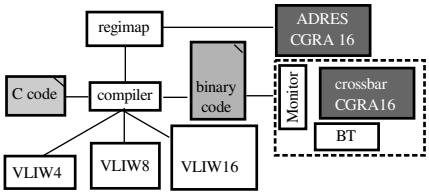

Fig. 10. Target Platforms: 4 compiled-based and the proposed BT CGRA

The proposed MS-BT algorithm is suitable for RISC binary code as well as VLIW code. During the BT execution, the RAW vector is updated after all single instructions inside a VLIW instruction are processed.

V. Experimental Results

As proof of concept, the proposed MS-BT algorithm is evaluated by using a cycle-accurate simulator in comparison to off-line compilers, where we considered a 1-issue VLIW as baseline MIPS-like processor capable to execute one instruction per clock cycle. Moreover, a comparison to an ADRES-based CGRA [19] is performed by using the state-of-the-art modulo scheduling tools [10]. Fig. 10 depicts the experimental framework used to measure the performance of 5 target architectures. The proposed run-time approach is compared to 4 static compiler-based approaches: three VLIW processors and one CGRA-based approach. The VLIW-n is an n-issue processor, and the C code is compiled by using the option -o3 (basic loop unrolling and trace scheduling compilation) and the option -o5 (very heavy loop unrolling) [18]. The binary code of VLIW-4 is the starting point for the proposed MS-BT mechanism. As mentioned before, it can also be applied to another ISA as MIPS-like code.

The VLIW processors and the proposed CGRA are evaluated under two distributions of heterogeneous units. Both distributions use up to 2 multipliers, and *n* ALUs per clock. The difference between them lies in the number of memory units: 1 or 2, as memory latency and bandwidth is a critical resource nowadays. In addition, the inner loop dataflow is extracted and the modulo scheduling algorithm REGImap [10] is used to map the loops onto an ADRES reconfigurable architecture. We chose the REGImap approach because it is the state-of-the-art for compiler-based approach to find optimal scheduling solutions. Additionally, REGImap can use up to 8 local registers and a set of homogeneous units (the current REGImap version supports only homogeneous units). Regarding the ADRES architecture, although it has a mesh topology, which has less routing resources compared to the proposed heterogeneous crossbar CGRA, the evaluated ADRES architecture is homogeneous, and hence, there is no placement constraint due to the unit type. Moreover, there is no constraint in the maximum number of operations per clock cycle (memory, multipliers or ALU). Although the interconnection is more restricted, there is no placement constraints or limit on the maximum number of a critical resource per clock cycle. The proposed target CGRA has 16 heterogeneous units: M memory units, 2 multipliers and 14-M ALUs, where M is the number of memory units (1 or 2). The units are interconnected by a crossbar network.

The increment vector example and four multimedia bench-

TABLE I. NUMBER OF INSTRUCTIONS PER LOOP

Loop	Inst	Ld	St	M	A	ILP1	ILP2
cjpegt1.txt	78	8	8	13	32	4.87	9.75
cjpegt2.txt	79	8	8	13	33	4.93	9.87
matrixt1.txt	56	16	0	17	21	3.50	7.00
x264t1.txt	52	12	0	7	13	4.33	8.67
itvert1.txt	108	7	4	25	30	8.64	8.64
itvert2.txt	63	8	8	5	22	3.94	7.88
itvert3.txt	100	6	4	25	26	4.00	4.00
itvert4.txt	66	10	10	5	30	3.3	6.6
itvert5.txt	55	4	2	13	14	8.46	8.46
itvert6.txt	60	5	2	13	16	8.57	9.23
itvert7.txt	63	8	8	5	22	3.94	7.88

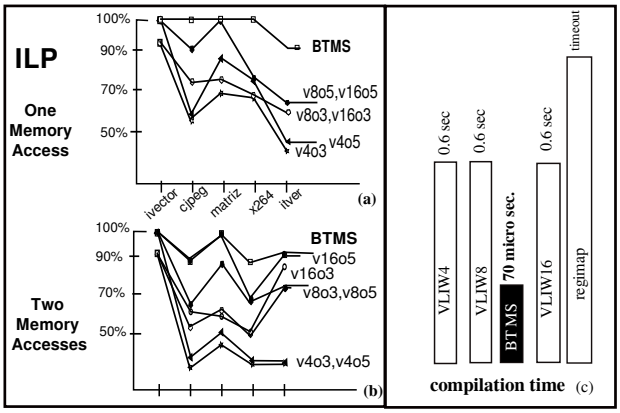

Fig. 11. Normalized ILP for 1 or 2 memory access per clock cycle

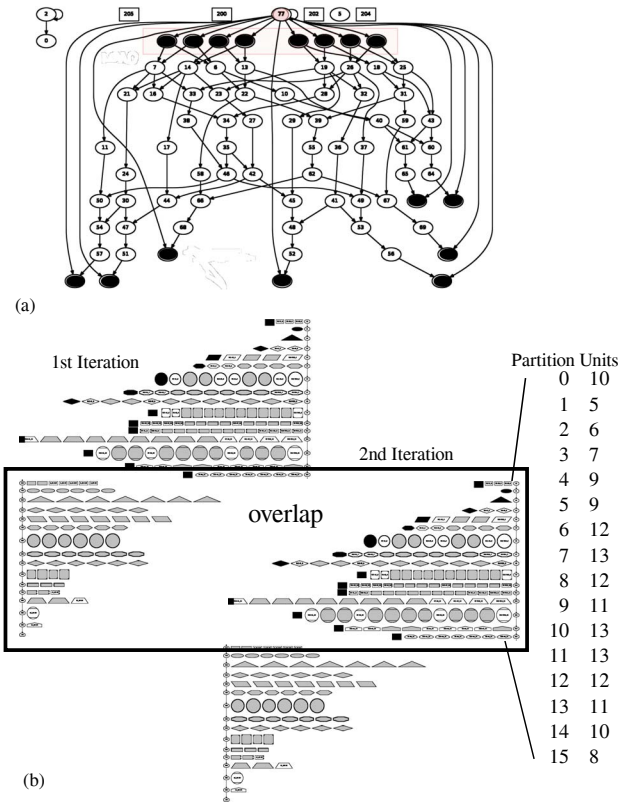

Fig. 12. (a) DFG DCT Row's Loops (b) Mapped two overlap iterations

marks have been used to validate our approach. Table I presents the instruction composition of the detected loops from the binary. The first column shows the benchmark name and a number, when there is more than one inner loop. Some loops are omitted as they have the same instruction composition. The number of MIPS instructions is presented in column Inst, followed by the number of load, store, multiplications, and ALU instructions, respectively. Columns ILP1 and ILP2 present the maximum theoretical ILP bound by memory throughput of 1 or 2 accesses per cycle, respectively.

The experiments described next were performed to verify the quality of the scheduling to reach the maximum ILP available and the required compiler and/or execution time. The VLIW code has been compiled for 4, 8, and 16-issue with -o3 and -o5 options [18], and 1 or 2 memory accesses per clock cycle. For all approaches, the ILP was measured by considering only the inner loop code and normalized by the maximum theoretical ILP depicted in Table I. The ADRES results were generated by using the REGImap algorithm with 8 local registers and 16 units. Even though the dataflow graphs have a medium size from 50 to 120 operations, REGImap could not find a scheduling solution for most of them in less than 1 hour. REGImap can only map the single increment vector example in 2 seconds. However, the graph has been modified by using one local index counter adder for each load-add-store chain to eliminate the fanout of the index counter. The same strategy was applied to the cjpeg loop which has 16 load/store instructions controlled by the R_2 address register. Instead of one address register, the cjpeg was modified to use four registers. REGImap has found a scheduling after 4 hours. As one can observe, the proposed approach is 3 orders of magnitude faster than the VLIW static compiler solution as shown in Fig. 11. The VLIW results are labelled by V_iO_j, where i is the VLIW issue and j the compiler option. Concerning the quality of the scheduling measured by the ILP, the MS-BT reaches the optimal solution in 4 of 5 benchmarks for 1 memory access per cycle. Moreover, the reached ILP is better than the ILP found by the VLIW processors even with 16-issue. For 2 memory accesses per clock cycles, the ILP achieved by the MS-BT approach is quasi-optimal, and the average ILP is 92.5% of the maximum theoretical ILP.

The next experiment analyzes a loop from DCT (discrete cosine transform) application [20] onto the CGRA. This algorithm implements an 8x8 DCT in two steps: rows and columns. In the example detailed next, we considered the loop to process the rows. The dataflow graph has been extracted from the VLIW binary code, and it is depicted in Fig. 12(a). The source code has been compiled to a 4-issue VLIW (1 memory, 2 multiplier, and/or 4 ALU).

Based on maximum ILP, the MS-BT will map the binary code by using modulo scheduling to overlap the loop iterations. The goal is to produce an overlap as depicted in Fig. 12(b), where the maximum number of units per partition is equal or less than N, where N is the maximum number of units available per partition in the target architecture. The black vertices represent the scheduling memory operations. There is only one memory operation per clock cycle, which shows that the memory constraints are not violated. The buffer registers are displayed by using grey vertices and the latency is 31 clock cycles. As can be observed, the worst cases of unit usage are configurations 7, 10, and 11, where 13 units are required. It is

TABLE II. ARCHITECTURE AREA AND FREQUENCY EVALUATION

Architecture	BRAMs	LUTs	Clock
VLIW 4	16	6575	91 Mhz
Crossbar CGRA	23	12977	103 Mhz
ADRES	4	15173	92 Mhz
VLIW 8	64	17490	62 Mhz

also important to take into account the maximum number of live variables (or registers). In a MIPS processor or in a VLIW processor, the maximum number of registers is bounded by the register file size, which, in general, it is 32 or 64. For the proposed approach, each functional unit has two input registers to store the live variables. Therefore, for our 16 unit CGRA, there are 32 registers.

Assuming a VGA image with 640x480 pixels, the complete DCT application runs in 24.5ms (or 2,457,600 clock cycles) in a soft-core 100MHz 4-issue VLIW. On the other hand, when including the accelerator and the binary translation overhead, the execution time reduces to 15ms at 100MHz. Furthermore, a theoretical analysis also demonstrates a great potential to increase speedup. If we assume an aggressive scaling that enables an ILP with of 8.88 by allowing 2 memory accesses per clock cycle, the execution time reduces to 7ms. Additionally, for a 5 Mega-pixel image, which is a normal size nowadays, the binary translation overhead is insignificant.

Finally, all five target architectures have been implemented on a commercial FPGA (Xilinx XC6VLX240T-1FFG1156) synthesized with ISE version 13.3 to evaluate the relative performance and area. The ADRES implementation is based on the architecture described in [15] with a homogeneous set of functional units. Additionally, to provide a consistent comparison, all architectures use the same functional units: ADRES, CGRA, and the VLIW processor [21]. The functional units support the execution of all VLIW instructions.

Table II presents the results in amount of resources and maximum operation frequency after the placement and routing steps. The number of LUT slices and BRAM are depicted. It is important to notice that the VLIW processor uses BRAM to implement the register file. This is one of the most expensive resources in a VLIW architecture, since connections to all the functional units must be provided, which makes the size of register file (RF) grows exponentially. For instance, in VLIW-16, the RF should allow 32 reads and 16 writes at same time. In addition, the VLIW16 has a fully interconnected network to implement the forward logic. For this reason, VLIW-16 occupies the entire FPGA and it is not depicted in Table II. On the other hand, the area cost of the proposed CGRA16 is lower than the VLIW8 and the ADRES16 architectures. Additionally, the CGRA16 clock frequency is faster than ADRES16 and the VLIW's frequencies. However, since our architecture tightly couples a VLIW processor and a CGRA, the total area is the sum of the both. Considering the 4-issue VLIW, the total area of our architecture is equivalent to a standalone 8-issue VLIW.

VI. RELATED WORK

In this work, the proposed solution consists of a binary translation mechanism applied to the modulo scheduling to translate code from different ISA sources into a CGRA. Concerning MS approaches, the ADRES/DRESC compiler proposed by Mei, et al., [15] is one of the first works to propose the use of modulo scheduling to accelerate inner loops in a CGRA tightly coupled to a VLIW processor. The DRESC presents an algorithm that combines modulo scheduling, simulated annealing and pathfinder techniques for scheduling, placement and routing, respectively. The experimental results demonstrate that a long time is required to perform scheduling, in order of minutes for a 64-FU reconfigurable architecture.

In an attempt to reduce compile time, Park, et al., [4] proposed a modulo scheduling mechanism targeted to CGRA that prioritizes routing and performs placement and scheduling during routing process. According to the authors, routing is a very time-consuming step in CGRAs. Therefore, by focusing on an efficient routing algorithm, it is possible to map dataflow graphs to the CGRA faster than the solutions that performs routing after scheduling, which in turn, reduces compile time. The results presented in [4] demonstrate shorter compile time when compared to DRESC. However, EMS presents a lower scheduling quality than DRESC, which results in performance penalty during execution time.

To improve mapping quality, the approach in [2] proposed the combination of routing and recomputation in the mapping algorithm, called EPIMap. The EPIMap heuristic transforms an input graph to an epimorphic equivalent graph that meets all the CGRA constraints. Additionally, the algorithm performs a systematic search of the solution space, which ensures a higher quality mapping. According to the results presented in [2], EPIMap achieves near-optimal mapping quality, with longer compile time when compared to [4]. Moreover, a formal model and NP-completeness proof for the modulo scheduling CGRA is presented in [2].

In [10], EPIMap's authors proposed a solution that allows a better usage of local register files by the mapping algorithm in order to increase performance, called REGIMap. The local register files are used to temporarily store data that will be used in next cycles. In this case, it is not necessary to hold the current value in the processing element (PE), since the value is stored in the local register. Therefore, the PE can be scheduled to start a new operation with no risk of overwriting the old value that will still be used in another cycle. In [10], the authors present a simple example that demonstrates the advantage of this solution. Additionally, REGIMap mapping algorithm performs scheduling and placement in different steps (routing is performed during scheduling and placement steps). According to the authors, comparison results between REGIMap and DRESC [3] show performance increase of 1.89x and 56x lower compilation time. In spite of that, the experimental results presented in [10], for a 4x4 mesh CGRA and different number of local registers, are still in order of thousands of seconds (against milliseconds of our approach, with similar quality).

The ADRES framework [15] is also a design exploration tool where several interconnection patterns, local register files, and functional unit resources have been investigated. The mesh-plus interconnection reaches better tradeoffs since each CGRA unit connects to any other unit in at most two steps. By enriching the interconnection model, it is possible to reduce the routing complexity. However, the reduction in the compile time can come at the price of more interconnection area. Recent works proposed full interconnection model to

simplify the placement and routing [22], [7], [23]. Even for a fully interconnected architecture, the modulo scheduling is still an NP-complete problem, as proved in [2] for a generic interconnection model.

From all the mentioned solutions, only the one proposed in [9] presents low compile time and can be moved from compile time to run-time. In spite of that, this solution still requires a compiler to analyze application and generate the graphs. Therefore, due to this compiler dependence, they are only suitable for Just In Time (JIT) compilation. Our binary translation approach is the first module scheduling algorithm starting from binary code, in order to ensure software compatibility between different ISAs. Therefore, the solution presented in this paper proposes a novel binary translation mechanism that has a mapping algorithm that is completely compiler free. In this mechanism, loop detection, graph generation, modulo scheduling, placement and routing are performed during execution time.

VII. Conclusions

This paper presented the first binary translation approach for the modulo scheduling algorithm in CGRAs. Additionally, our approach is completely adaptable to the amount of resources. In the results, it was demonstrated the efficiency of the same mechanism when the amount of memory elements was increased. To evaluate area, quality and execution time of the scheduling algorithm, and performance gains, we presented a set of experiments comparing the proposed solution with two other systems, the VLIW processor (4-, 8- and 16-issue) and the REGImap/ADRES modulo scheduling solution, which is currently, the most efficient modulo scheduling compiler-based approach. In spite of that, since the loop graphs has index counters with a large fanout, REGImap was not able to manage it. Even considering the fact that REGImap is based on EPImap, which proposes recomputation to handle fanout greater than 3. Concerning area occupancy, the comparisons demonstrated that the proposed CGRA with 16 functional units (CGRA16) plus a 4-issue VLIW is equivalent to an 8-issue VLIW. Quality analysis results have shown that the proposed run-time mechanism with CGRA16 presented the quasi-optimal ILP better than the VLIW solution (4-, 8- and 16-issue). Compile time results have also demonstrated that while the other solutions (VLIW and ADRES) required a minimum of 0.5 seconds to compile, the proposed mechanism required 100 microseconds when running on a commercial processor (Intel i5) and only 3 milliseconds when running in a softcore processor. These gains are around 3 orders of magnitude, enabling the use of mechanism for run-time mapping. An example of the proposed binary translation mechanism mapping a loop from DCT application to a generated RA presented a speedup factor of 1.8 when compared to the same loop running in a VLIW.

Finally, the proposed BT approach is flexible and adaptable, enabling its extension to take into account other program phases than inner loops as well as its implementation in hardware by using a FSM. Future works include evaluating the acceleration considering larger application blocks and conditional branches and on-the-fly generation of the CGRA using customized functional units.

References

[1] Z. Salcic, D. Hui, P. S. Roop, and M. Biglari-Abhari, "Hidraa reactive multiprocessor architecture for heterogeneous embedded systems," *Microprocessors and Microsystems*, vol. 30, no. 2, pp. 72 – 85, 2006.

[2] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Design Automation Conference*, 2012, pp. 1280 –1287.

[3] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: a retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. FPT*, 2002, pp. 166 – 173.

[4] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. PACT*, 2008.

[5] B. De Sutter, P. Coene, T. Vander Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in *Proc. LCTES*, 2008, pp. 151–160.

[6] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. LCTES*, 2009, pp. 21–30.

[7] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro, "An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Proc. CASES*, 2011.

[8] L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," in *Proc. FPT*, 2012.

[9] R. Ferreira, V. Duarte, W. Meireles, M. Pereira, L. Carro, and S. Wong, "A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures," in *SAMOS XIII*, 2013.

[10] M. Hamzeh, A. Shrivastava, and S. B. Vrudhula, "Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras)." in *Design Automation Conference*, 2013, p. 18.

[11] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '01, 2001.

[12] J. K. Paek, K. Choi, and J. Lee, "Binary acceleration using coarse-grained reconfigurable architecture," *SIGARCH Comput. Archit. News*, vol. 38, no. 4, pp. 33–39, Jan. 2011.

[13] J. Bispo, N. Paulino, J. Ferreira, and J. Cardoso, "Transparent trace-based binary acceleration for reconfigurable hw/sw systems," *Industrial Informatics, IEEE Transactions on*, 2012.

[14] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proc. MICRO*, 2009, pp. 370–380.

[15] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proc. DATE*, 2003.

[16] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proc. MICRO*, 1994, pp. 63–74.

[17] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2008, pp. 1208–1213.

[18] H. Labs. (2013) Vex toolchain. [Online]. Available: http://www.hpl.hp.com/downloads/vex/

[19] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the adres coarse-grained reconfigurable array," in *Proc. ARC*, 2007, pp. 1–13.

[20] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-d dct algorithms with 11 multiplications," in *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*. IEEE, 1989, pp. 988–991.

[21] S. Wong, T. Van As, and G. Brown, "ρ-vex: A reconfigurable and extensible softcore vliw processor," in *International Conference on Field-Programmable Technology FPT*. IEEE, 2008, pp. 369–372.

[22] M. Shami and A. Hemani, "Morphable dpu: Smart and efficient data path for signal processing applications," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, 2009, pp. 167–172.

[23] L. Zhou, H. Liu, and J. Zhang, "Loop acceleration by cluster-based cgra," *IEICE Electronics Express*, vol. 10, no. 16, 2013.