J Sign Process Syst
DOI 10.1007/s11265-015-0974-8

A Dynamic Modulo Scheduling with Binary Translation: Loop Optimization with Software Compatibility

Ricardo Ferreira · Waldir Denver · Monica Pereira · Stephan Wong · Carlos A. Lisbôa · Luigi Carro

Received: 20 October 2014 / Revised: 26 December 2014 / Accepted: 21 January 2015

Abstract In the past years, many works have demonstrated the applicability of Coarse-Grained Reconfigurable Array (CGRA) accelerators to optimize loops by using software pipelining approaches. They are proven to be effective in reducing the total execution time of multimedia and signal processing applications. However, the run-time reconfigurability of CGRAs is hampered overheads introduced by the needed translation and mapping steps. In this work, we present a novel run-time translation technique for the modulo scheduling approach that can convert binary code on-the-fly to run on a CGRA. We propose a greedy approach, since the modulo scheduling for CGRA is an NP-complete problem. In addition to read-after-write dependencies, the dynamic modulo scheduling faces new challenges, such as register insertion to solve recurrence dependences and to balance the pipelining paths. Our results demonstrate that the greedy run-time algorithm can reach a near-optimal ILP rate, better than an off-line compiler approach for a 16-issue VLIW processor. 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. In this analysis it is demonstrated that when changing from one memory access per cycle to two memory accesses per cycle, the modulo scheduling algorithm is able to exploit this increase in memory bandwidth and enhance performance accordingly. Additionally, to measure area and performance, the proposed CGRA was prototyped on an FPGA. The area comparisons show that a crossbar CGRA (with 16 processing elements and including an 4-issue VLIW host processor) is only 1.11 × bigger than a standalone 8-issue VLIW softcore processor.

Keywords Modulo scheduling · Binary translation · Run-time · Coarse-grained reconfigurable accelerator

R. Ferreira (✉) · W. Denver
Universidade Federal de Viçosa, Vicosa, Minas Gerais, Brazil
e-mail: ricardo@ufv.br

M. Pereira
Universidade Federal do Rio Grande do Norte,
Natal, Rio Grande do Norte, Brazil

S. Wong
TU Delft, Delft, Netherlands

C. A. Lisbôa · L. Carro
Universidade Federal do Rio Grande do Sul,
Porto Alegre, Rio Grande do Sul, Brazil

1 Introduction

In the past decade, a large effort was spent on enhancing the performance of multimedia and signal processing applications. The improved capabilities of system-on-chip designs triggered by the rapid increased popularity of embedded systems led to increased complexity and size of these applications. Such (computationally intensive) applications are usually characterized by intensive loops and common use of arrays. Different architectural design solutions [6] were proposed to increase performance and reduce power consumption. These solutions include DSPs, GPUs, VLIWs, and ASIPs combined with a myriad of techniques, e.g., loop transformations, software pipelining, and compiler optimizations [2, 24, 26].

While off-the-shelf architectures, such as VLIWs, DSPs, and GPUs are general solutions designed to meet many constraints and still work for a wide range of applications, ASIPs are normally targeted for a small set of applications. In this sense, ASIPs are capable of achieving higher speedups than off-the-shelf solutions. However, in

Published online: 17 February 2015

the embedded systems market, with many new applications with different behavior emerging at a high pace, ASIPs are not able to provide enough speedup.

Coarse-grained reconurable architectures are good candidates to cope with such a challenge, since they can provide both power efficiency and hardware acceleration [14], as well as flexibility to adapt to emerging applications. Additionally, they have a lower reconfiguration overhead than fine-grained reconfigurable architectures, such as FPGAs [17]. Many proposed solutions aiming to increase performance during the execution of loops, using Modulo Scheduling and Coarse-grained reconfigurable architectures (CGRAs), can be found in the literature [8–11, 14, 16, 27, 29, 32]. In [33], the authors highlight the amount of nested loops in multimedia applications that can be parallelized by CGRAs through the use of software pipelining or other techniques. In spite of that, all those solutions require special compilers or modifications in the application, which, in turn, precludes software compatibility and code reuse.

The use of compile-time techniques is mainly caused by the complexity of the data dependence graph extraction and the mapping algorithm. Mapping instructions onto the CGRA includes placement, routing, and scheduling. During these steps, the mapping algorithm has to take into account resource limitations - when the amount of parallel instruction is higher than the amount of processing elements - and data dependences among instructions. This last constraint is normally solved through the use of the data dependence graph or data-flow graph (DFG). Therefore, the compiler generates the DFGs of the application and then, the mapping uses it to perform the other steps.

More recently, some works proposed the use of binary translation to provide software compatibilty for CGRAs [5, 30]. Binary translation converts code compiled from a source ISA to run in a different ISA. This can be used to enable application execution in different ISAs without the need for code recompilation. This is useful for CGRAs since the code compiled to a host processor should be translated into a configuration of the CGRA.

In order to reduce mapping complexity and, at the same time, meet the requirements of code reuse and software compatibility, we propose a novel binary translation (BT) mechanism for a run-time modulo scheduling (MS) algorithm. The MS algorithm maps inner loops onto a CGRA accelerator. To the best of our knowledge, this is the first work to propose a BT run-time modulo scheduling algorithm for CGRAs. The proposed BT MS algorithm reduces mapping time by proposing alternatives to many of the complex solutions presented in previous modulo scheduling algorithms [8, 9, 11, 14, 27, 29, 32]:

1. Eliminating the need for intermediate DFG generation: In this work, we propose a novel algorithm to detect, generate, and schedule the loop directly from the binary code.
2. Using a greedy placement step: Since the modulo scheduling for CGRA is an NP-complete problem, as proved in [14], the proposed modulo scheduling uses a greedy algorithm to find the local optimal solution and scheduling time [10, 11].
3. Using a crossbar as interconnection among processing elements: Similar to the MS-JIT approach [10], A CGRA with a crossbar network reduces complexity, when compared to the widely used mesh topologies [14, 28, 32, 33]. A crossbar-based CGRA is used. Nevertheless, the MS-JIT assumes that the starting point to perform the MS is a DFG, and therefore it requires special JIT compilers or modifications in the applications (such as pragmas) to detect the loop and to generate the DFG.
4. Using local register files connected to processing elements: Reduces the overhead of accessing centralized general-purpose register files. This solution is also used in other approaches [8, 9, 16, 20, 31, 32].
5. Scaling process: For instance, if the memory bandwidth is increased, the binary translator can easily incorporate this information in order to accelerate applications.

The proposed BT MS is compared to off-line VLIW compiler-based approaches and the REGIMap, an MS algorithm proposed in [16]. A set of inner loops from multimedia applications are used to measure the instruction level paralelism (ILP). The experimental results show that, although the BT MS is a greedy approach, it reaches an ILP very close to the optimal value. Furthermore, even though BT MS is executed in software as trap routine, the overhead it imposes is very low. Moreover, the proposed architecture has been prototyped in a FPGA, and the area and clock latency evaluated and compared to the VLIW and mesh-based CGRA approaches.

The remainder of this paper is organized as follows. Section 2 explains the modulo scheduling technique, presents some basic concepts and compares the proposed technique with GPU and VLIW solutions. Section 3 details the proposed CGRA architecture. In Section 4, we present the binary translation modulo scheduling (BT MS) algorithm. Experimental results are discussed in Section 5. Section 6 examines some works related to the proposed solution. Finally, Section 7 presents the conclusions and future works.

2 Modulo Scheduling

This section details the modulo scheduling (MS) approach. Section 2.1 introduces the MS approach by using a simple

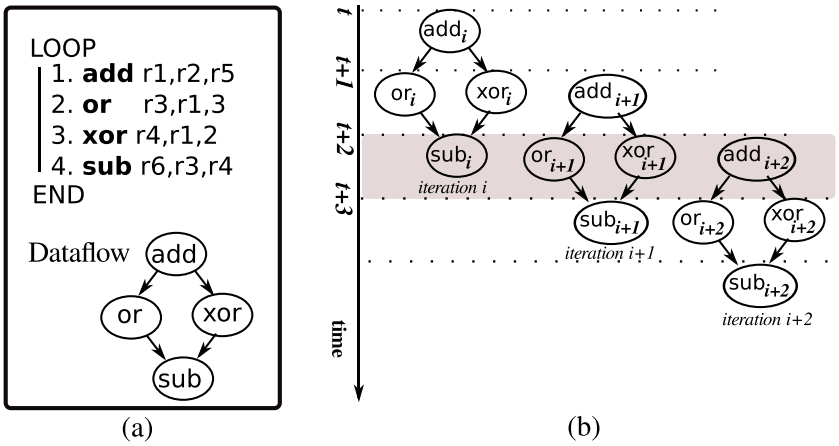


Figure 1 (**a**) A loop code and its DFG; (**b**) Three overlaped iteractions.

example. Subsequently, Section 2.2 formally introduces MS concepts and definitions. Finally, Section 2.3 compares the MS approach to three traditional approaches: Tomasulo in superscalar architecture, graphical processing units (GPUs), and VLIW.

2.1 Simple Example

Modulo scheduling (MS) [34] 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 simple 4-instruction loop code depicted in Fig. 1a and its DFG. Since there are read-after-write register dependencies (RAW) between instructions, the ADD instruction precedes the following instructions (OR and XOR), and then the last instruction (SUB) depends on the previous two instructions. In this example, only the OR and XOR instructions could be executed in parallel. Therefore, using conventional ILP exploitation, each iteration needs at least three clock cycles to be executed. In order to increase performance, modulo scheduling overlaps iterations, thereby reducing the amount of clock cycles required to execute the loop. To demonstrate that, suppose that a new loop iteration is started at every clock cycle, as depicted in Fig. 1b. As shown in the shaded area of the Fig. 1b, at time $t + 2$, three iterations $i, i + 1$, and $i + 2$ are overlapped, and four instructions (maximum ILP) are being executed in parallel: SUB_i, OR_{i+1}, XOR_{i+1}, and ADD_{i+2}. In this scenario, four instructions are executed per cycle, and at every clock cycle one loop iteration is completed.

When modulo scheduling is applied to a CGRA, the algorithm must include the placement of instructions by mapping each one to a functional unit. MS must take into consideration the resource limitations and the maximum ILP that can be achieved. In most solutions that combine MS with CGRA, a compiler extracts the DFG and, then, the MS maps the DFG into the architecture. Once again referring to the example depicted in Fig. 1, in order to exemplify this next step in the MS algorithm, let us assume the use of a 2x2 mesh-based architecture, with four functional units (FU), as depicted in Fig. 2a. A valid scheduling at time $t+2$ is depicted in Fig. 2b, where there are instructions from the three iterations being executed in parallel. An MS algorithm should map the instructions in time (scheduling) and space (placement). In addition, the placement should perform a valid routing in order to comply with data dependencies. As an example, instruction ADD_{i+2} (placed in FU_3) will send the r_1 value to OR_{i+2} and XOR_{i+2}, which will be executed at time $t + 3$ as depicted in Fig. 2c.

An MS algorithm should also be able to map a DFG larger than the target architecture. Let us assume a target architecture that consists of only two functional units, as depicted in Fig. 3a. An MS algorithm could generate a valid mapping, such as the one depicted in Fig. 3b. Since there are only two functional units and four operations must

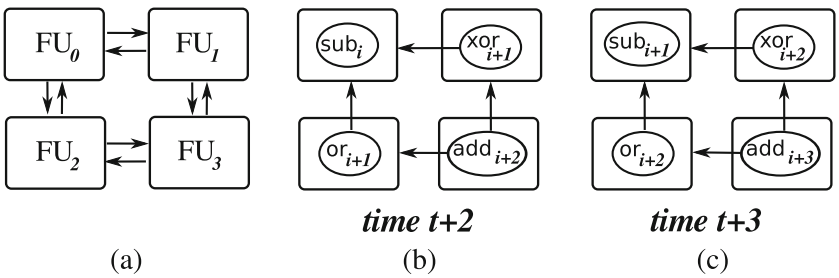


Figure 2 (**a**) 2x2 Mesh Architecture; (**b**) Scheduling at Time $t + 2$; (**c**) Scheduling at Time $t + 3$.

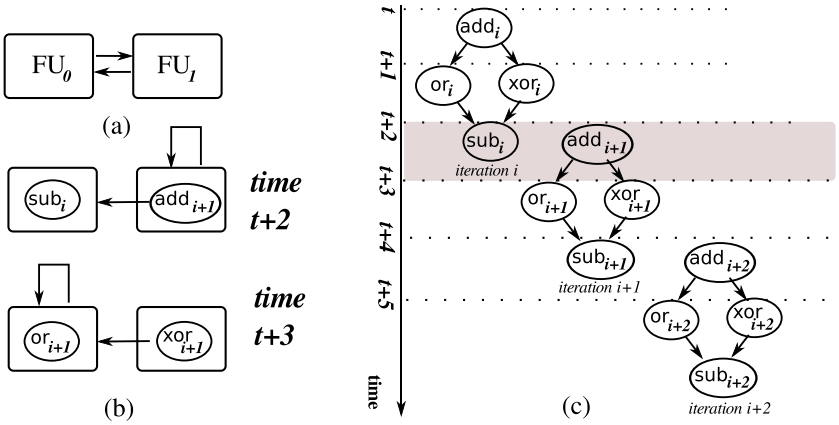


Figure 3 (**a**) 2 unit architecture; (b) Scheduling at time $t+2$ and $t+3$; (**c**) Iteration overlapping.

be performed, two temporal partitions are used and only two iterations will overlap, as shown in Fig. 3c. As an example, at time $t+2$, FU_0 executes the last instruction of iteration i and FU_1 computes and sends the value of r_1 from iteration $i+1$. At time $t+3$, FU_0 executes the instruction OR_{i+1} using the r_1 value generated during the previous clock cycle, and forwards the new value of r_3, while FU_1 computes and sends the new value of r_4. The computed values of r_3 and r_4 will be forwarded and used in the next clock cycle by FU_0, which will compute the last instruction (SUB_{i+1}) of iteration $i+1$ at time $t+4$.

Once the scheduling is computed, the architecture will be configured to execute the loop. The 2-unit architecture detailed in Fig. 4a consists of a set of processing elements (PEs), an interconnection network, and a configuration memory. Each PE comprises one functional unit (FU) and local registers that store temporary values. Figure 4b depicts the configuration memory contents and scheduling for the example shown in Fig. 3b. The MS algorithm generates a loop scheduling, which is repeated at regular intervals. The loop is executed by incrementing the address register. In this example, as shown in Fig. 3b, a new loop iteration is started every two cycles. Therefore, the scheduling interval or initialization interval (II) is 2, and the next address is computed by $address = (address + 1) \bmod II = (address + 1) \bmod 2$.

2.2 Basic Concepts

More formally, the MS algorithm maps a dataflow graph onto a Time Extended Architecture, or a TEC graph, as defined in [14]. Figure 5b depicts a TEC graph for the 2×2 mesh architecture depicted in Fig. 5a. The number of temporal partitions $P_0, P_1, \ldots, P_{II-1}$ (or temporal dimensions) is equal to the initialization interval (II). TEC is a graph generated by unrolling in time the target architecture. The TEC representation of the architecture shows all interconnections between two consecutive temporal partitions. The MS algorithm performs a spatial and temporal mapping onto the TEC. A connection between two PEs in the architecture will lead to a TEC interconnection between all consecutive temporal partitions. For instance, $PE_0 \rightarrow PE_1$ will generate the connections $PE_0^{Pi} \rightarrow PE_1^{Pj}$ for all i where $j = i+1 \bmod II$, and mod is the modulo operator. The last partition P_{II-1} is connected to the first partition P_0, since this is a modulo scheduling algorithm. Moreover, if the PEs have internal registers, all PEs have self-connections between the partitions, that is $PE_k^{Pi} \rightarrow PE_k^{Pj}$ where $j = i+1$ MOD II, as shown by the horizontal arrows in Fig. 5b.

Figure 6 depicts a DFG for a loop with eight instructions, where each instruction is represented by a number. II is computed by dividing the number of instructions by the

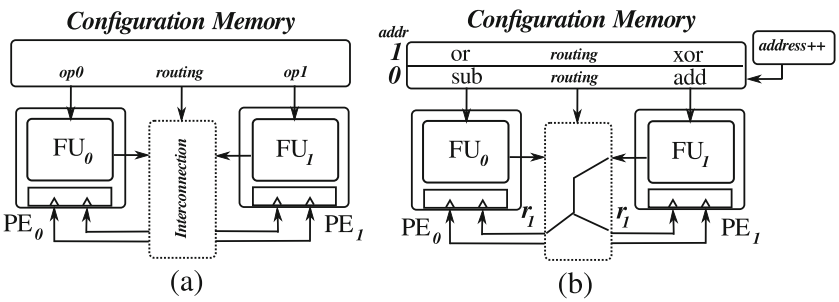


Figure 4 (**a**) Detailed target architecture; (**b**) A valid scheduling.

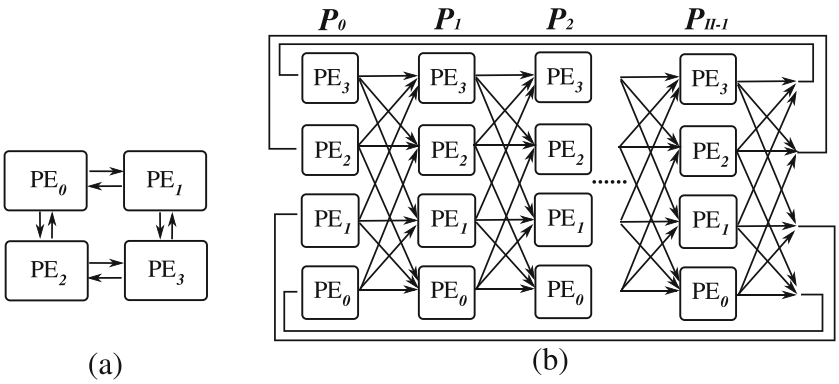


Figure 5 (**a**) Target architecture; (**b**) Time extended architecture (TEC) graph.

number of processing elements (PEs). Since the architecture has 4 PEs, the value of II should be at least 2, which means that a new iteration can only be started at least two cycles after starting the previous one. The MS algorithm will map the DFG onto the TEC_2 graph as shown in Fig. 6. Let us assume that there is a data dependence between two instructions: $x \rightarrow y$. In this case, if x is placed in partition P_i, y should be placed in the next partition, P_{i+1}. For instance, node 1 is connected to nodes 2 and 3, therefore if node 1 is assigned to $PE_0^{P_0}$, nodes 2 and 3 should be placed in an adjacent PE in partition P_1. Figure 7a depicts one possible partial scheduling, where nodes 2 and 3 are placed in $PE_0^{P_1}$ and $PE_1^{P_1}$, respectively. Since nodes 2 and 3 have been placed in partition P_1, their successors should be placed in P_0, as depicted in Fig. 7b. Although the TEC has a total of eight PEs, there are four PEs in each partition. Therefore, the MS fails, since node 7 is placed in P_1 and there is no free PE in P_0 to map its descendent node 8 (see Fig. 7c).

The MS algorithm works similarly to that of the bin package problem, where the node scheduled at time t_i is placed in the partition (bin) P_k, where $k = i \bmod II$. For instance, the nodes at time t_0, t_2, and t_4 will be mapped onto the bin P_0 (see Fig. 6). As already mentioned, in this case, MS fails, because there are only four PEs in the bin P_0, for five nodes (1, 4, 5, 6, and 8). When it is not possible to perform the MS with the minimal number of partitions, the DFG should be re-scheduled or local registers could be used. Recently, the EPImap and REGIMap algorithms [14, 16] proposed the use of re-computation and local registers to find the minimal sets. 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 later to node 8, placed at $PE_2^{P_1}$. If this is not feasible with the minimal set, then the MS algorithm increases the partition (bin) number or the II until a solution is found. However, as expected, when II increases, the throughput and the ILP are reduced.

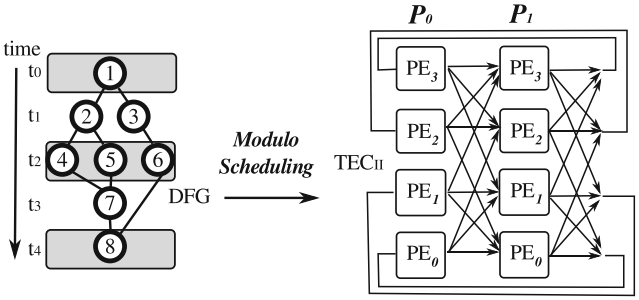


Figure 6 Modulo Scheduling as a graph mapping: $DFG \rightarrow TEC_{II}$.

Besides avoiding partition overflow, the MS algorithm must take into account other constraints. Assuming the DFG example depicted in Fig. 6a, the DFG paths should be balanced, since the execution is performed in a pipelined fashion. With this purpose, a buffer node could be inserted in edge $6 \rightarrow 8$ (see Fig. 8a). In addition, concerning memory operations, which impose the most severe constraints on performance, it is assumed that nodes 4, 7, and 8 are memory operations and the architecture can perform only one memory access per cycle. Therefore, at least three partitions are needed, as shown in Fig. 8b. Two loop iterations will be executed at the same time, in a pipelined fashion (see Fig. 8c). Every three clock cycles, a new iteration is started. The scheduling quality is measured by the minimal number of partitions, which corresponds to the maximum throughput. In this example, the solution is optimal and it is bounded by memory constraints. In summary, the MS algorithm must consider three main constraints: resources limitations, to avoid partition overflow; balanced paths, to temporally store values used in different temporal partitions; and memory constraints, due to the limited amount of parallel memory accesses.

2.3 Tomasulo, GPU, and VLIW

In this section, we compare the MS algorithm to three classical approaches: a superscalar processor, a VLIW processor, and a Graphics Processing Unit. First, let us consider a two-way superscalar processor with dynamic scheduling using Tomasulo algorithm. Let us suppose there are four functional units: 1 load/store, 1 multiplier, and 2 ALUs. Figure 9a depicts a loop code example. The superscalar

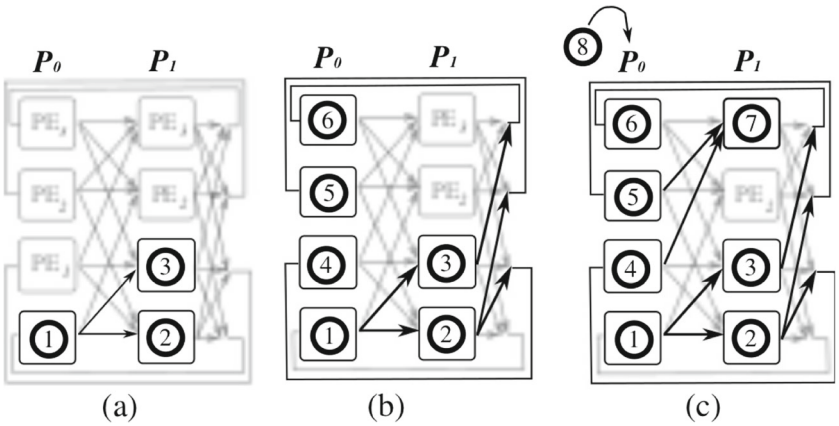


Figure 7 (**a**) Place 1,2, and 3; (**b**) Place 4, 5 and 6; (**c**) Fail to place 8 in P_0.

processor will fetch two instructions per cycle. Since there are read-after-write dependencies, the code will be executed as shown in Fig. 9b. The code is fetched, decoded and scheduled in every execution of the loop, which consumes energy. On the other hand, the MS algorithm would schedule only once, the configuration will be stored in a small local configuration memory, and the execution will be performed by overlapping two iterations, as depicted in Fig. 9c. The performance of the superscalar processor could be improved by applying compiler-based static scheduling approaches, such as loop unrolling. However, the static scheduling would be optimized for a specific superscalar processor, while the dynamic MS algorithm proposed in this paper is able to generate a new optimized scheduling on-the-fly. Thus, in case the target architecture is modified, a new scheduling is generated without the need for any offline modification. Additionally, even in case there are forwarding connections in the superscalar processor, the temporary register values are read and written from/to the global register file. For this example, the achieved ILP in the superscalar processor is 1.75, which is close to the maximum $ILP_{two-way} = 2$. It is important to notice that, in spite of the achieved ILP being close to the theoretical maximum, the superscalar has 4 functional units. In contrast, the CGRA has also four processing elements and the MS algorithm achieved an ILP of 3.75, which is also very close to the maximum theoretical $ILP_{MS} = 4$.

The second classical approach is a VLIW processor and its corresponding compiler. The VLIW simplifies the architecture and transfers the burden of dependence checking and scheduling to the compiler. For the purpose of this comparison, we assume a 4-issue VLIW, as depicted in Fig. 10a, and suppose the loop is unrolled twice. Figure 10b depicts a scheduling where the even/odd iterations are represented by using white and black foregrounds in the boxes representing the instructions. In this example, the VLIW achieves an ILP of 2.8. Again, it is important to highlight that the VLIW is a 4-issue unit. In order to improve ILP, the compiler should apply more aggressive unrolling. Moreover, the multi-ported global register file (GRF) is one of the most power hungry parts in any VLIW processor. The MS algorithm proposed in this work does not require any global register file, and, in addition, it is dynamic in comparison to the static compiler-based VLIW approach.

The GPU is the third approach to be compared and it is also compiler-based. The first drawback in solutions using GPUs is the fact that the source code (C, C++, etc.) should be modified when targeting GPU/CPU architecture. In this

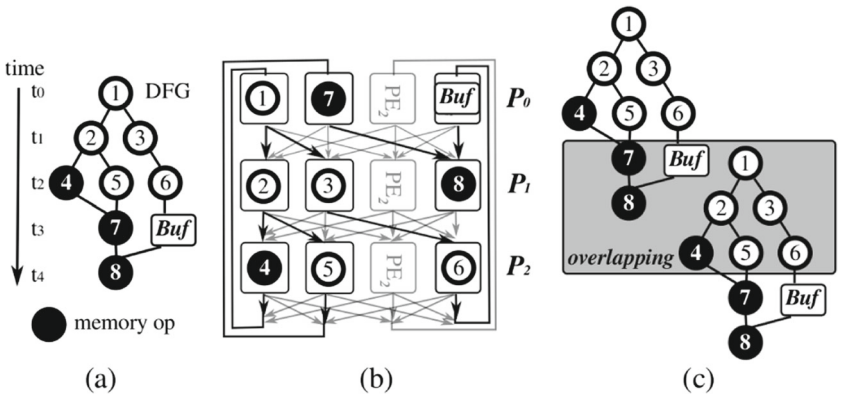


Figure 8 (**a**) Buffer insertion to balance the DFG; (**b**) TEC_3 graph; (**c**) Resulting loop overlapping.

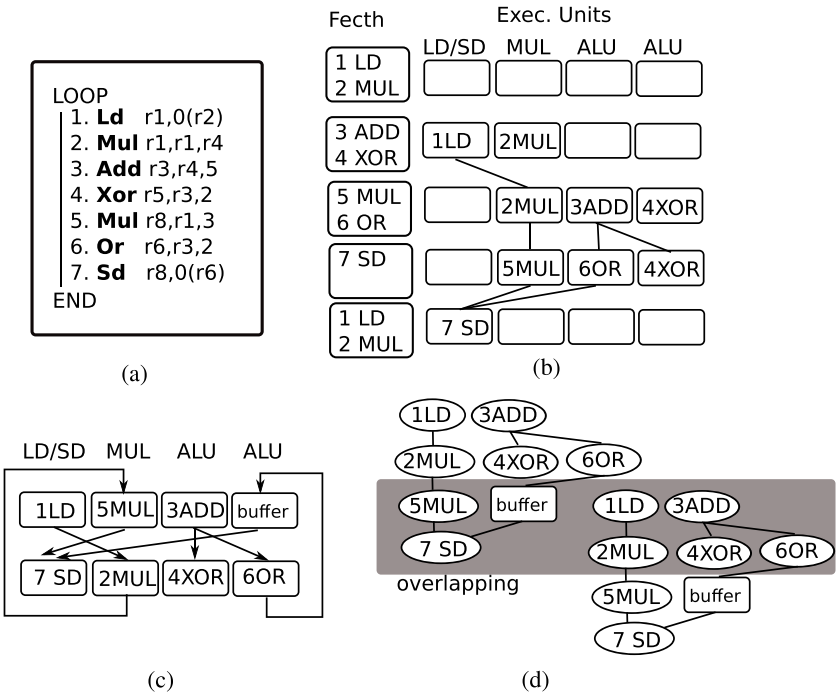


Figure 9 Tomasulo *versus* MS approach: (**a**) Loop code; (**b**) Two-way superscalar execution (**c**) Modulo Scheduling (**d**) Resulting MS loop overlapping.

comparison, it will be considered a four-thread GPU, as depicted in Fig. 11. Each of the four units of the GPU is able to execute any operation (memory access, multiplication and ALU operations). Each thread will execute the same operation, in a SIMD fashion, and four loop iterations are executed in parallel. In this example, the GPU achieves the maximum ILP of 4. However, it requires more powerful units in comparison to the previous approaches, and it also requires more registers, since four threads are executed simultaneously.

In all three analyses, the classical solutions present significant drawbacks, when compared to the modulo scheduling CGRA. In the comparisons all architectures have been considered with the same amount of processing elements. In spite of that, the superscalar and VLIW were not able to achieve the same ILP achieved by MS. The VLIW could increase the ILP, at the cost of a more optimized compiler, and the GPU required much more powerful processing units and more registers. On the other hand, the modulo scheduling CGRA is able to achieve or get closer to the theoretical ILP, with the same amount of processing elements, and a run-time scheduling. This demonstrates how the proposed solution is competitive in a scenario where classical architectures are used to increase performance of loop-based applications.

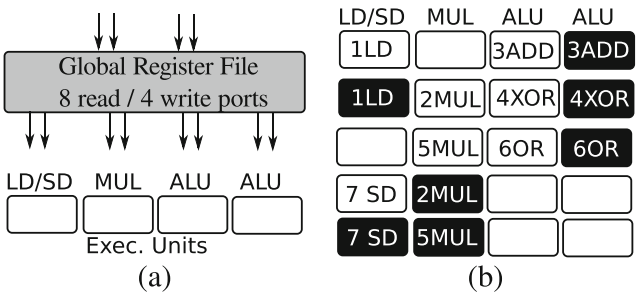


Figure 10 VLIW: (**a**) Functional Units; (**b**) Unrolling twice.

3 Architecture

The modulo scheduling approach was initially proposed by [34] and the modulo scheduling targeting coarse-grained reconfigurable architecture (CGRA) was introduced by [27]. Figure 12a shows a general view of a CGRA, which consists of three components: a set of processing elements (PEs), an interconnection network, and a configuration memory. During the last decade, several MS algorithms have been proposed, and most of them use mesh or mesh-plus topologies. The processing elements can be homogeneous [9, 12, 14, 18, 23, 27, 29, 31] or heterogeneous [1, 8, 21, 32, 36]. However, the MS algorithms

Thread 0	Thread 1	Thread 2	Thread 3
Id r2,**thread.id**	**Id** r2,**thread.id**	**Id** r2,**thread.id**	**Id** r2,**thread.id**
Ld r1,0(r2)	**Ld** r1,0(r2)	**Ld** r1,0(r2)	**Ld** r1,0(r2)
Mul r1,r1,r4	**Mul** r1,r1,r4	**Mul** r1,r1,r4	**Mul** r1,r1,r4
Add r3,r4,5	**Add** r3,r4,5	**Add** r3,r4,5	**Add** r3,r4,5
Xor r5,r3,2	**Xor** r5,r3,2	**Xor** r5,r3,2	**Xor** r5,r3,2
Mul r8,r1,3	**Mul** r8,r1,3	**Mul** r8,r1,3	**Mul** r8,r1,3
Or r6,r3,2	**Or** r6,r3,2	**Or** r6,r3,2	**Or** r6,r3,2
Sd r8,0(r6)	**Sd** r8,0(r6)	**Sd** r8,0(r6)	**Sd** r8,0(r6)

Figure 11 A four thread GPU approach.

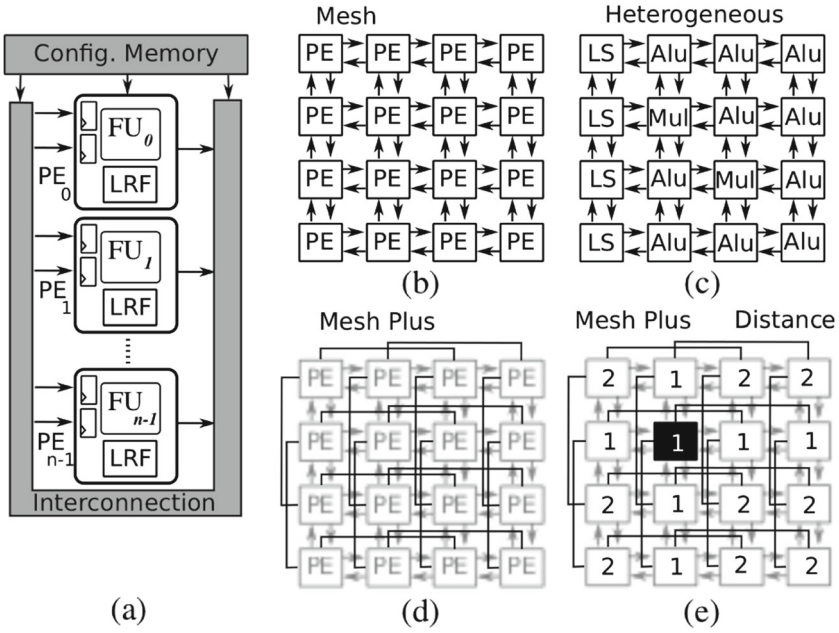


Figure 12 CGRA architectures: (**a**) Generic architecture (**b**) Mesh-based archicteture; (**c**) Heterogeneous mesh; (**d**) Mesh-plus; (**e**) Mesh-plus disntance.

proposed in those previous works are highly time consuming, due to the selected scheduling approach, as well as due to their slow placement and routing steps (P&R).

The P&R complexity can be reduced by increasing the connectivity among processing elements. One way to do that is adding more wires, for instance, as in the mesh-plus with one-hop connections depicted in (Fig. 12d). The one-hop connections reduce the distance between the PEs, as depicted in Fig. 12e, where the numbers indicate the distance of each PE from the black PE. In a 4×4 mesh, 7 PEs have distance 1 and 9 PEs have distance 2. As already mentioned in Section 2.2, it is important to notice that the MS algorithm uses a temporal/spatial mapping, and each PE has a connection to itself, used forward its computed value to the next temporal partition of the TEC graph. Therefore, these numbers represent a temporal/spatial distance.

Another way to simplify the P&R is to use a homogeneous set of PEs, since a heterogeneous one imposes more constraints on the placement. Moreover increasing the connectivity or the local PE capability reduces the P&R time, and many previously proposed solutions used this strategy, those previous MS algorithms are still too time-consuming to be implemented in a run-time approach. Therefore, more investments in connections and in the MS algorithm should be done.

Recently, a Just-in-Time approach for the MS algorithm has been proposed in [10]. The MS-JIT algorithm applies two strategies to reduce the mapping time. Firstly, the target architecture is a crossbar-based CGRA, in order to reduce the complexity of the placement and routing steps. According to [10], the complexity reduces from NP-complete to $O(1)$. However, the scheduling step itself is still NP-complete [10]. Secondly, the MS algorithm is implemented by using a greedy approach based on graph traversal, where each node is visited only once. However, the MS-JIT approach [10] does not include the DFG generation, as depicted in Fig. 13a, and an off-line compiler is required to generate the DFG, similarly to other MS approaches.

The approach proposed in this work significantly improves the previous one [10] by eliminating the DFG generation. It starts from the binary code and requires neither changes in legacy source code nor any compiler support. Its main advantage is to provide a compiler independent solution, which eliminates the need for special compilers and the re-compilation time required to map instructions, as opposed to previously described solutions. The proposed architecture is also simplified by using a heterogeneous CGRA instead of the homogeneous one proposed in [10]. In order to reduce the time required to exchange data between the host processor and the CGRA, a tightly coupled accelerator approach is proposed, as shown in Fig. 13b. The host processor can have a RISC or a VLIW instruction set architecture (ISA). The CGRA copies the values from the processor register file to the CGRA inputs, then the loop body is executed, and, finally, the output values are written back to the processor register file. A simple monitor module detects a jump instruction: while the first loop iteration is executed by the processor, the monitor concurrently verifies whether all loop instructions are suitable to translation. In case the loop is a candidate to be executed in the CGRA, the program execution is interrupted, and the processor executes the binary translation routine to generate the CGRA

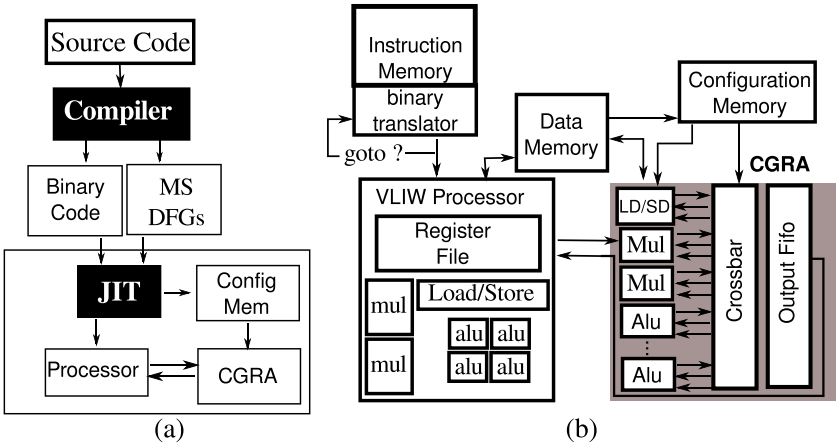


Figure 13 (**a**) Previous work, a JIT approach [10]; (**b**) Proposed Architecture.

configuration on-the-fly. From then on, the CGRA accelerator will execute all the remaining loop iterations.

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. The approach can be extended to detect/execute more complex loop structures.

The target CGRA is a heterogeneous architecture interconnected by a crossbar network. As already mentioned, a heterogeneous PE set reduces the area cost and the amount of configuration bits. In this work, we assume the use of three PE types: ALUs, multipliers, and load/store units. Moreover, the architecture has 16 PEs in total, since the throughput does not increase very much beyond the size of 16, as demonstrated in [33]. In addition, a crossbar with 16 units is more feasible, when compared to 4x4 mesh-based architectures.

Figure 14 depicts the internal structure of a processing element PE. Each PE has two input registers, R_a and R_b, where the computation from the previous clock cycle is stored. These registers are also used as buffer registers (BR), as explained in [10], and, in this case, the PE is bypassed and it cannot be assigned to any instruction in a given temporal partition. While REGIMap uses the local register file (LRF) in parallel to the local ALU, in the approach introduced here the LRF only stores immediate operands and loop input values. The goal is to simplify the MS algorithm to make it suitable for execution at run-time. Moreover, the use of input registers and/or a local register file present in each PE avoids the overhead imposed by the centralized multi-ported register files present in VLIW processors, which in turn, reduces the dynamic power consumption [13, 19]. Furthermore, our proposed CGRA provides a larger amount of parallel operations with a simpler input register. The CGRA allows 32 concurrent register reads and 16 register writes, distributed in its 16 PEs, as illustrated in Fig. 14. All live values are stored in these temporary input registers, avoiding the execution of reads/writes operations from/to the centralized register file during the whole loop execution.

The modulo scheduling algorithm was implemented based on the architecture described above. The main features that must be taken into account by the algorithm are the crossbar interconnection model, the local register files, and the heterogeneous processing elements. The main challenge is to generate a scheduling scheme that benefits from all these features in an attempt to maximize ILP and, at the

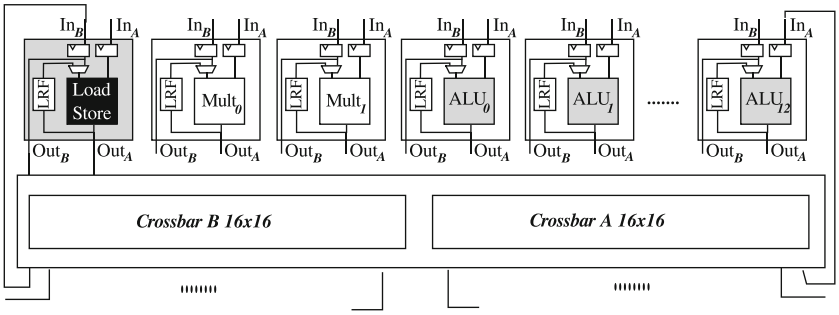


Figure 14 Detailed CGRA architecture.

same time, it is fast enough to run at execution time. The algorithm is described in the next section. It is important to highlight that, since the MS algorithm is a run-time solution, modifications in the architecture, such as amount and type of processing elements, can be managed as input parameters to the algorithm, without the need for modifications, since an array is used to keep track of the number of units being used during each time interval and another one is used to keep track of unit types.

4 Binary Translation Modulo Scheduling

This section details the BT MS algorithm. First, Section 4.1 introduces an example of an increment vector loop to illustrate a comparison of two approaches: an off-line VLIW compiler and the proposed run-time BT MS algorithm. Section 4.2 shows the BT MS pseudo-code and its features: RAW hazards, buffer insertion, heterogeneous units, and recurrence values. Finally, the ISA support is commented.

4.1 Increment Vector Example

As already discussed in Section 2, the MS approach differs from the classical Tomasulo, VLIW and GPU approaches. Concerning binary translation (BT), a mechanism to map binary MIPS code onto a CGRA, where a BT unit implements a dynamic Tomasulo-based algorithm customized for a heterogeneous CGRA, was proposed in [3]. All code blocks can be mapped to a large CGRA, and a configuration cache is used to store the most common blocks. However, the BT [3] does not use software pipelining. For instance, considering an inner loop with 32 instructions mapped by the BT proposed in [3]. Let us suppose that the latency for this loop is 8 cycles, and therefore the achieved ILP will be $32/8 = 4$. In order to achieve a high degree of parallelism, the CGRA [3] must have at least 32 units, and, in this solution, these units are not used in a pipelined fashion. On the other hand, our MS algorithm overlaps loop iterations. Let us consider an MS CGRA with 16 units and assume that there is a feasible scheduling with two temporal partitions. In this case, every two cycles a new iteration is processed, and the achieved ILP is $32/2 = 16$. Therefore, the CGRA size is reduced by a factor of 2 (16 units instead of 32 units), and the performance is improved 4× by the MS approach, when compared to BT [3] for inner loop acceleration.

In order to allow inner loop acceleration, besides detecting the read-after-write hazards, the BT MS also has to detect the recurrence values. The recurrence constraints are caused by loop-carried dependencies between the iterations. In addition, the BT MS should take into account the temporal partitions, in order to balance the pipeline paths. Aiming to provide the correct balance, the BT MS inserts buffer registers when it detects an unbalanced path. Nevertheless, in case the schedule fails, the BT MS increases the number of partitions, thereby affecting the overall performance. As one can see, the BT MS proposed in this work differs significantly from the previous superscalar Tomasulo and CGRA BT approaches proposed in [3] to dynamically solve new challenges. Furthermore, all previous MS approaches are compiler-based and time consuming, while the approach introduced in this paper is the first binary translation based one for the MS algorithm.

Before describing the BT MS approach in detail, a simple loop example is presented here. While in Section 2 the code examples were selected for illustrative purposes, in order to highlight some features of the MS algorithm, the following code example was generated by a compiler. Initially, this example was compiled considering only one memory access operation per clock cycle. Subsequently, our BT dynamic approach is compared to a VLIW compiler-based approach. Finally, the capability to perform more memory operations

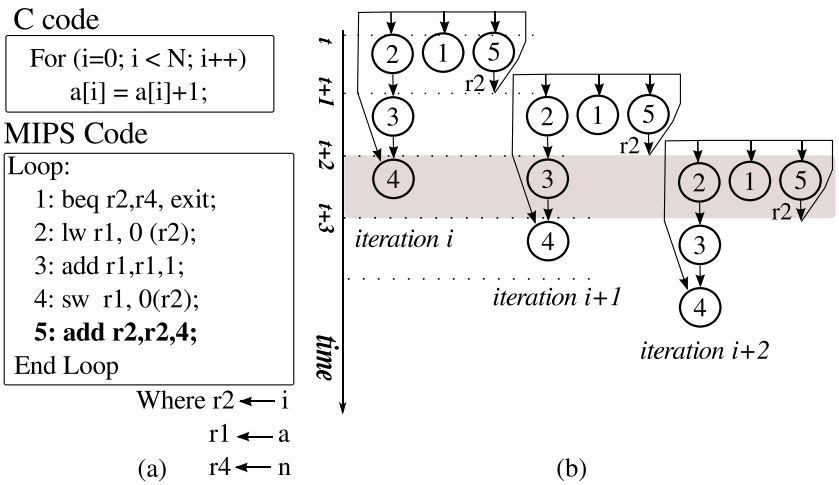


Figure 15 Simple Vector Increment: (**a**) Source code and MIPs code; (**b**) Dataflow.

per clock is used to show the adaptability of our approach, when compared to static compiler-based approaches. In the example, a simple loop is used to increment the values of the elements of one vector. The C code and the corresponding pseudo MIPS code are depicted in Fig. 15a. Figure 15b shows the data flow graph for an ideal case of a software pipelining execution. For ease of explanation, the instructions are numbered according to their order in the source code. The recurrence dependence on the index variable i, implemented by using register r_2, creates a DFG feedback edge. By using a CGRA with 5 units or more, the execution of one loop iteration can be completed every clock cycle, as can be observed from time $t + 2$ on, with five instructions being executed in parallel.

It is well-known that loop unrolling is a common approach that allows VLIW processors to reach a high ILP for inner loops. Figure 16a depicts the assembly code generated by a VLIW compiler [22] for a 4-issue VLIW processor. The compiler uses an unroll factor of 4. The code has 8 VLIW instructions. Each instruction can have up to 4 operations (grouped inside each dashed block). Each vector element should be read (load) and written (store). Thus, with an unroll factor of 4, at least 8 instructions are needed, since only one memory access operation per clock cycle is allowed (black box) and four instructions are needed to perform the load operations and another four are needed to perform the store operation. In this example, the compiler reaches the optimal ILP, and one element of the vector is added every two clock cycles. Figure 16b depicts the DFG. There are 4 load-add-store dependence chains. For instance, one chain is $2_{load} \rightarrow 5_{add} \rightarrow 10_{store}$, which uses the register r_3 to hold the temporary value. One can observe that the *add* instruction is scheduled for two clock cycles later than the *load*. Suppose also that a memory operation is executed in two clock cycles. In addition, there are two recurrence constraints: registers r_2, which is the memory index, and r_4, which is the loop counter.

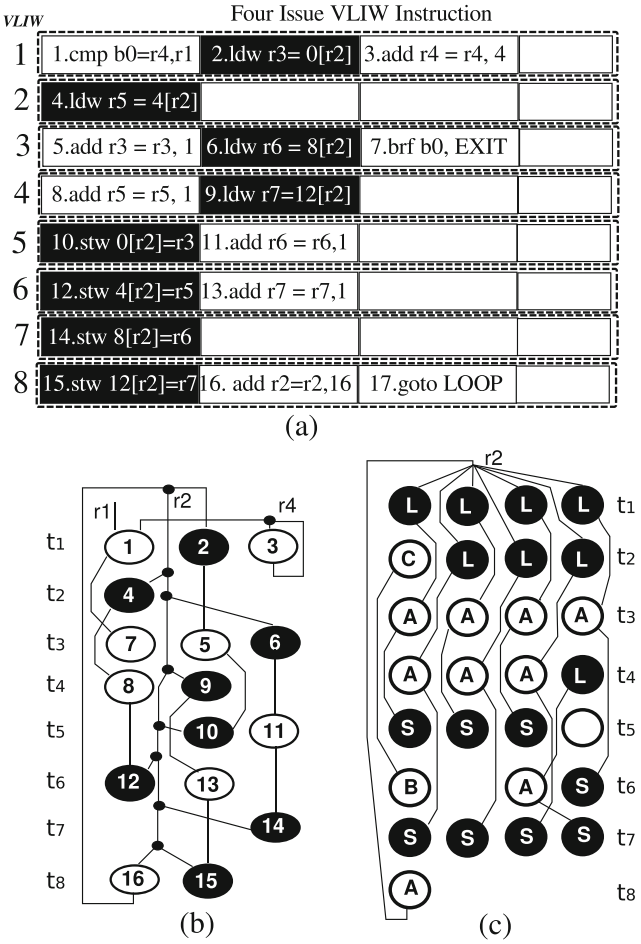


Figure 16 Increment Vector: (**a**) 4-issue VLIW code; (**b**) One Load/Store per cycle DFG; (**c**) Unrolling 8, 4 Load/Store 4-issue VLIW.

Now, let us assume that, thanks to some technology improvement, the architecture is able to 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. Figure 16c depicts the generated code for 4-issue and 4 memory ops per clock cycle. The code has also 8 VLIW instructions, however the compiler has applied an unroll 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. We can observe that the compiler fills almost all VLIW instruction slots. However, since the architecture is able to perform 4 memory ops per clock cycle, taking advantage of this fact only it is still possible to improve it to produce one loop result per $\frac{1}{2}$ cycle. In spite of that, the VLIW is limited to one result per cycle. Therefore, the VLIW compiled solution reaches 50 % of the performance of the optimal solution.

The BT MS mechanism proposed here scans the binary code, instruction by instruction, and performs scheduling dynamically. Let us consider the previous example for an architecture which supports 4 memory accesses per clock cycle. Also, let us assume that the input binary code is the 4-issue VLIW code depicted in Fig. 16a, which was originally compiled for one memory operation per cycle. Since there are 8 memory instructions, at least 2 partitions will be needed. Let us assume, also, that the load/store (L/S) operations are executed in a two-stage pipeline unit, with the first stage computing the address and the second stage sending/receiving data to/from the memory.

The BT MS scheduling after processing the 2 first VLIW instructions is shown in Fig. 17b. For ease of explanation, let us adopt the term operation to refer to a VLIW instruction slot. In Fig. 17b, operations are numbered from 1 to 7. Load/store operations are highlighted by black

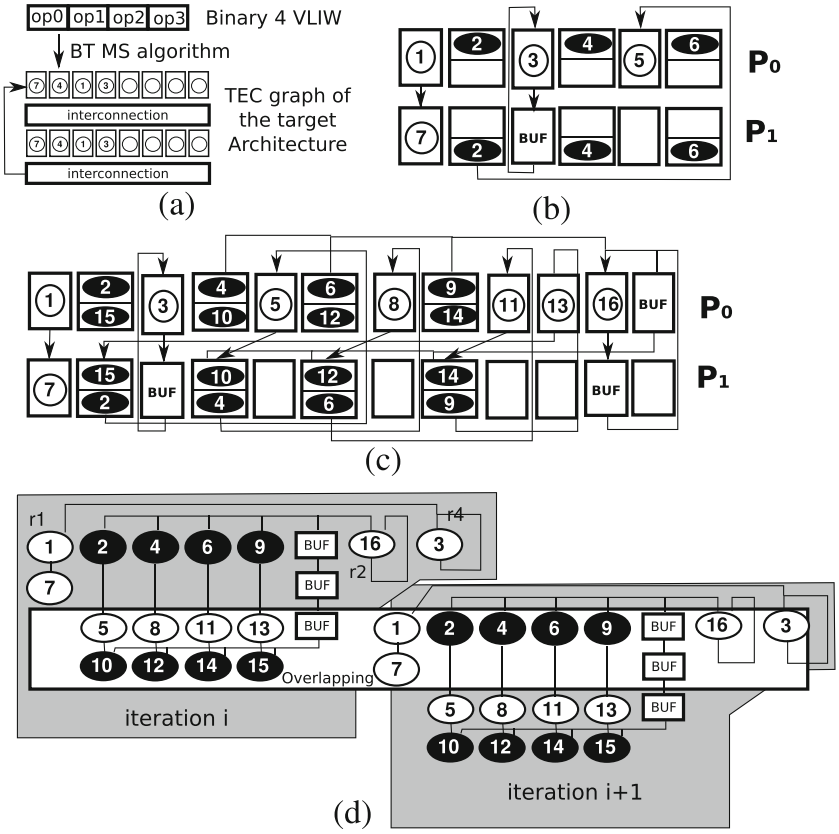


Figure 17 (**a**) Binary Translator Flow; (**b**) The TEC_2 graph after the two first vliw-instructions; (**c**) TEC_2 Final scheduling; (**d**) Loop Overlapping.

circles. Because memory access is performed in two-stage pipelines, as described before, these operations are inside a two-part box. The top part indicates that the operation is in the first stage (address computation) and the bottom part indicates that the operation is sending/receiving data. Operations 1, 2, 3 and 4 are scheduled for time t_0 in partition P_0, since there is no dependence. It is important to notice that, since operations 2 and 4 are load instructions, their results will be ready at time t_2 in partition $P_0 = (P_1 + 1) \bmod 2$. Add operation 5 is scheduled for time t_2 in P_0, since there is a RAW hazard due to r_3 produced by the load operation 2. Operation 6 is scheduled for time t_0, since there are free load slots and no dependence values, while operation 7 has a RAW due to the branch register b_0 produced at time t_0 by the *cmp* operation 1. Finally, operation 3 has a recurrence value and a buffer is added to forward r_4 across the temporal partitions.

Figure 17c depicts the final scheduling when using 12 PEs. We can observe the 4 load-add-store chains. For instance, the r_3 chain is $2_{load} \rightarrow 5_{add} \rightarrow 10_{store}$. In spite of a latency of 5 cycles to compute a load-add-store chain, the throughput obtained when using our approach is 2 clock cycles, due to the iteration overlapping, as depicted in Fig. 17d. Moreover, since 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.

4.2 BT MS Algorithm

Figure 18 depicts a pseudo-code of our 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.

Assuming one operand instruction. For instance, operation 2.*ldw r3=0[r2]* in Fig. 16a has one operand (algorithm lines 4–7). Since, it is the first time r_2 appears, it will be an input, and it will be inserted in the loop input register (LIR) list, which is detailed later. Furthermore, a load unit will be allocated at time t_0 in P_0, and the Write vector, which keeps track of all destination registers, stores register r_3. The time data adds the load delay, which is 2 for a memory operation. The second case is when there is a RAW dependence. For instance, the operation *5.add r3,r3,1* has a RAW in r_3, and in the algorithm, lines 9 and 10 are required to get the PE that computes r_3, as well as the time t_2 and the partition $P_i \bmod II$, where II is the initialization interval. Thus, a free unit is requested in this partition P_0, and the operation is placed and routed (line 11).

Moreover, the get_FreePE should handle another problem when there is no free PE in the target partition. For example, supposing the $12.Mult$ instruction as shown in Fig. 19a which has a RAW dependence in r_4. Suppose r_4

```
1   Inst = Fetch Instruction()
2   While ( Inst != END )
3     type = get InstructionType(inst), Partition = initial
4     if (Immediate Operand)
5       if (Rs1 ==  input)
6         PE = get_FreePE(type,Partition), Place  PE[partition] = Inst
7         LIR.insert(Rs1,partition,PE)  // added LIR list
8       else
9         PEraw = get PE[Rs1]      // Read-after-Write
10        p = partititon(PEraw),     PE = get_FreePE(type,p)
11        Place  PE[p] = Inst,       Route PERaw -> PE
12    else Two Value Operands:
13      if (Rs1 ==  input && Rs2 ==  input)
14        PE = get_FreePE(type,Partition), Place  PE[partition] = Inst
15        LIR.insert(Rs1,partition,PE)  // added LIR list
16        LIR.insert(Rs2,partition,PE)  // added LIR list
17      else   // Read-after-Write
18        PEraw = Later PE(rs1,rs2),      p = partititon(PEraw)
19        PE = get_FreePE(type,p),        Place  PE[p] = Inst
20        if (Rs1 == input ) LIR.insert(Rs1,p,PE)
21        if (Rs2 == input ) LIR.insert(Rs2,p,PE)
22        Route PERaw ->  PE,             Verify RCR(Target Register)
23    Inst = Fetch Instruction()
24  end While
25  Place and Route   LIR lists
```

Figure 18 Binary translation algorithm - Pseudo-code.

is generated by the *add* instruction allocated at PE_2 in P_1, and the $II = 3$. If there is no multiplier unit in P_2 to place 12.*Mult* as shown in Fig. 19b, buffer registers (BR) will be inserted until finding a free multiplier unit. For this example, one BR is inserted and 12.*Mult* is allocated in P_0.

Assuming two operand instructions (algorithm lines 13-22). There are three cases: (1) two inputs; (2) one input and one RAW; and (3) two RAWs. For instance, operation *10.stw 0[r2]=r3* in Fig. 16a has one input r_2 and one RAW in r_3. Operation 10 is allocated in partition P_1 after operation 5 due to the RAW, and buffer registers are allocated to balance the pipelining path from input r_2, as depicted in Fig. 16(c-d). Therefore, the BT MS applies buffer insertion and partition management, which is more complex than simple unit reference used to handle RAW dependence by the Tomasulo algorithm.

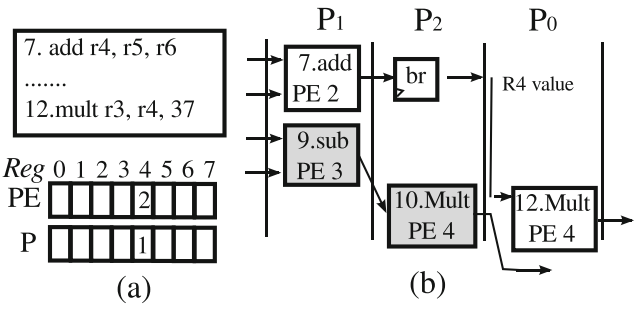


Figure 19 Buffer register insertion: (**a**) RAW code; (**b**) TEC_2 scheduling.

The BT MS should also 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. For instance, the r_1 in code example from Fig. 16a. Finally, there is the recurrence cycle register (RCR), which is similar to a LIR, however it is overwritten, for instance, r_2 and r_4 in Fig. 16a. The RCR can be a loop counter, vector index, or inter-iteration values.

Figure 20 presents an illustrative example to explain LIR and RCR management a simple loop. The code has a load-add-store chain through r_1 and an index vector r_2. 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, r_2 behaves as a LIR until the BT MS processes the last instruction. By default, all registers that appear as source registers are considered as a LIR, until they are overwritten and become an RCR. Moreover, there is register r_1 in Fig. 20, 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.

Let us assume, for ease of explanation, that the scheduling is performed by using 5 partitions (0 to 4), as depicted in Fig. 20b. The two-stage load is executed at PE_0 in P_0 and P_1. The result is sent to the adder at PE_1 in P_2, and finally it is sent to the two-stage store at PE_0 in P_3 and P_4. The adder at PE_1 is used in P_0 to execute instruction 4 (add r_2) and in P_2 to execute instruction 2 (add r_1), as the units are time-multiplexed, and Fig. 20b depicts the TEC graph as introduced in Section 2.2. 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 P_0, in PE_0 (load), and also in PE_0 (store) in P_3. Moreover, an additional 4 BR chain is generated to send back the value to the adder. Although the TEC graph in Fig. 20b depicts 7 BRs in total, the target architecture uses only 2 BRs, since the TEC graph is a time unrolling architecture, and the maximum number of BRs is the maximum number per partition. For this example, P_1 and P_2 use at most two BRs.

For this example, a better scheduling is possible. Assuming only one memory access operation per clock cycle, the minimal number of partitions is 2. Figure 20c depicts the mapping by using two partitions. Similar to the example depicted in Fig. 17c, the load is mapped in P_0 and P_1, then the adder in P_0, and finally the store in partitions P_1 and P_0, respectively. Every two clock cycles, the loop produces a new value. At resource level, the usage is maximum in P_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 r_2 LIR chain has also three BRs as the previous mapping with 5 partitions depicted in Fig. 20b. However, there is

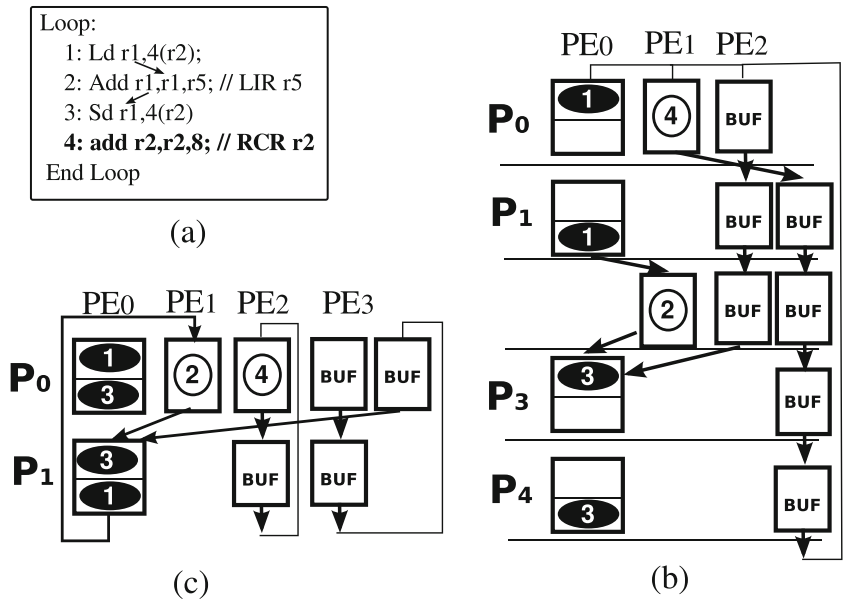


Figure 20 LIR and RCR registers: (**a**) Code (**b**) TEC_5 scheduling; (**c**) TEC_2 scheduling.

an overlapped iteration, and the BRs in P_0 store values of different iterations.

The proposed BT MS algorithm is suitable for different ISAs, such as RISC binary code as well as VLIW code. Regarding a VLIW code, the RAW vector is only updated after all operations inside a VLIW instruction are processed.

5 Experimental Results

As proof of concept, the proposed run-time BT MS algorithm is compared to 7 off-line compiler-based options as depicted in Fig. 21. Three target VLIW architectures are evaluated. For each one, two optimization options are applied. 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) [22]. The performance is evaluated by using a cycle-accurate simulator available in [22], where we considered a 1-issue VLIW as baseline MIPS-like processor capable to execute one instruction per clock cycle. We select the binary code of VLIW-4 as a starting point for the proposed BT MS mechanism, since it is the less optimized version. As mentioned before, it can also be applied to another ISA as MIPS-like code.

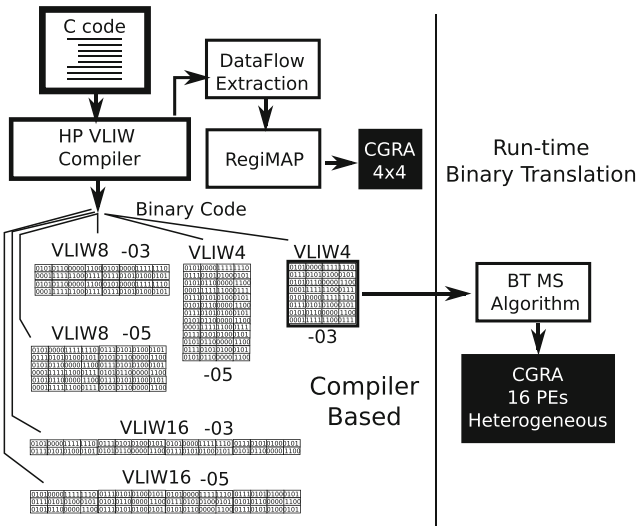


Figure 21 Target Platforms: 7 compiled-based options and our proposed BT approach.

Moreover, a comparison to MS compiler-based approach targeting the ADRES CGRA [7] is performed by using REGIMap [16]. 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 our 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).

The BT MS 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 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.

Table 1 presents the instruction composition of the detected inner-most loops from the binary of four multimedia benchmarks: Cjpeg, Itver2, MatMul, x264. The increment vector example from Section 4.1 is also evaluated. The first column in Table 1 lists the benchmark name and a loop ID number, when there is more than one inner loop. Some loops are omitted as they have the same instruction composition. The number of MIPS-equivalent 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 access per cycle, respectively.

The ADRES results with 8 local registers and 16 units were mapped by using REGIMap. 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. The drawback of REGIMap for these evaluated loops is due to the multiple-fanout of index counter registers as r_2 shown in Fig. 16b.

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.

Table 1 Innermost loops: Instruction distribution.

Loop	MIPS Inst	Type				Maximum	
		Ld	St	M	A	ILP1	ILP2
cjpegt1	78	8	8	13	32	4.87	9.75
cjpegt2	79	8	8	13	33	4.93	9.87
matrixt1	56	16	0	17	21	3.50	7.00
×264t1	52	12	0	7	13	4.33	8.67
itvert1	108	7	4	25	30	8.64	8.64
itvert2	63	8	8	5	22	3.94	7.88
itvert3	100	6	4	25	26	4.00	4.00
itvert4	66	10	10	5	30	3.3	6.6
itvert5	55	4	2	13	14	8.46	8.46
itvert6	60	5	2	13	16	8.57	9.23
itvert7	63	8	8	5	22	3.94	7.88

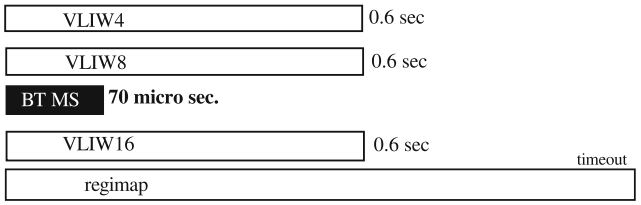


Figure 22 Average Compile Time *versus* run-time BT MS.

The VLIW code has been compiled for 4, 8 and 16-issue (see Fig. 21) with −o3 and −o5 options [22], and 1 or 2 memory access 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 1. Regarding amount of time required for compilation, referred here as compile-time, the BT MS proposed approach, in addition to be executed in run-time, it is 3 orders of magnitude faster than the VLIW static compiler solution as shown in Fig. 22.

Concerning the quality of the scheduling measured by the ILP, the results depicted in Fig. 23a shows that the BT MS reaches the optimal solution in 4 of 5 benchmarks for 1 memory access per cycle. Moreover, the achieved ILP is better than the ILP found by the VLIW processors, even with 16-issue. Figure 23b depicts the ILP when 2 memory accesses per clock cycles are allowed, the ILP of BT MS approach is quasi-optimal, even when compared to the VLIW16 −O5 compiler option. The BT MS average ILP is 92.5 % of the maximum theoretical ILP.

The next experiment analyzes a loop from the cjpeg application. This loop implements a DCT (discrete cosine

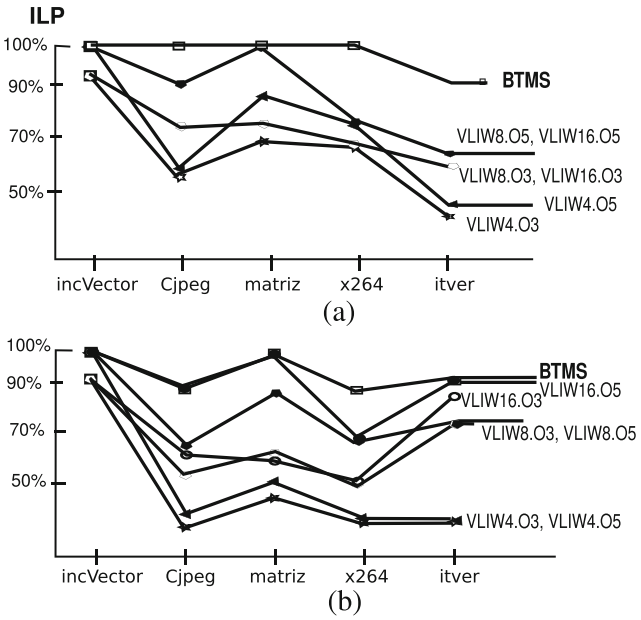


Figure 23 Normalized ILP: (**a**) One memory access per cycle; (**b**) Two memory accesses per cycle.

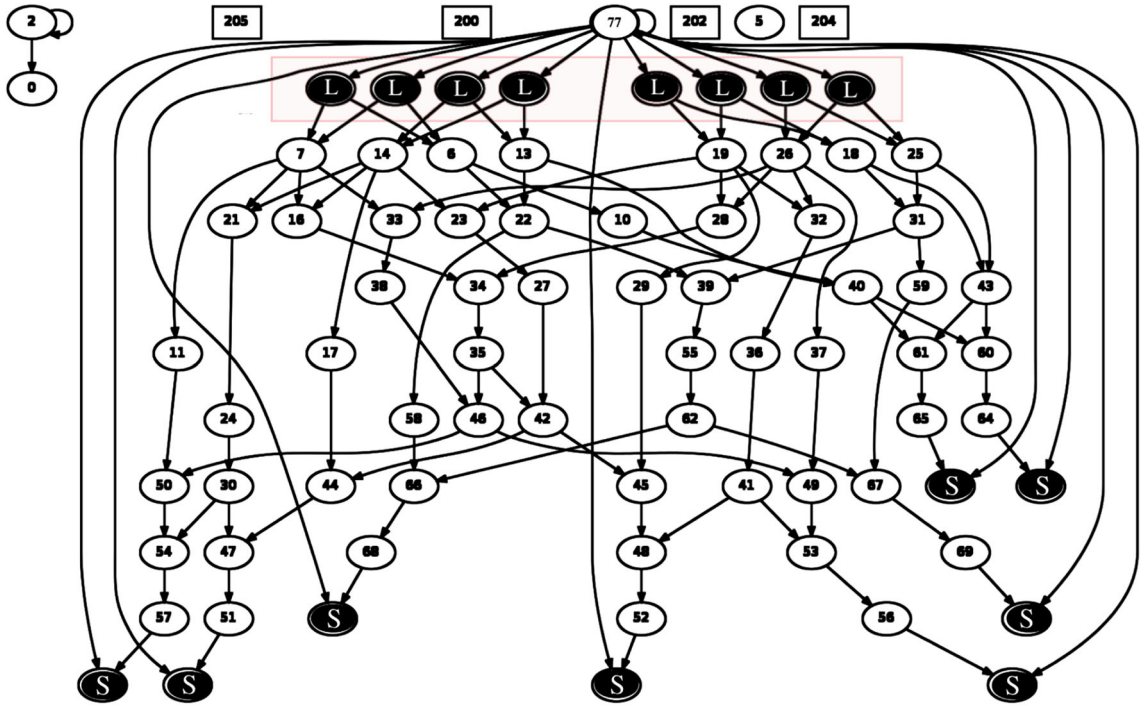


Figure 24 Cjpeg data flow graph has 78 instructions: 13 multipliers, 32 add/sub instructions, and 16 memory instructions.

transform) [25] onto the CGRA by using 8x8 DCT tiles in two steps: rows and columns. In the example detailed next, we considered the loop to process the rows. The dataflow graph (DFG) has been extracted from the VLIW binary code, and it is depicted in Fig. 24. The source code has been compiled to a 4-issue VLIW (1 memory, 2 multiplier, and/or 4 ALU). It is important to highlight that our approach does not require the DFG extraction as the previous JIT MS approach presented in [10]. The DFG is depicted only to identify the potential ILP. The DFG has 16 memory instructions (8 loads and 8 stores). Assuming one memory access per clock cycle, the II will be equal to 16 due to memory constraints.

Starting from the minimum II, the BT MS will map the binary code by using modulo scheduling to overlap the loop iterations. This example produces a two overlapping iteration execution. The TEC graph generated by the BT MS algorithm is depicted in Fig. 25, where the maximum

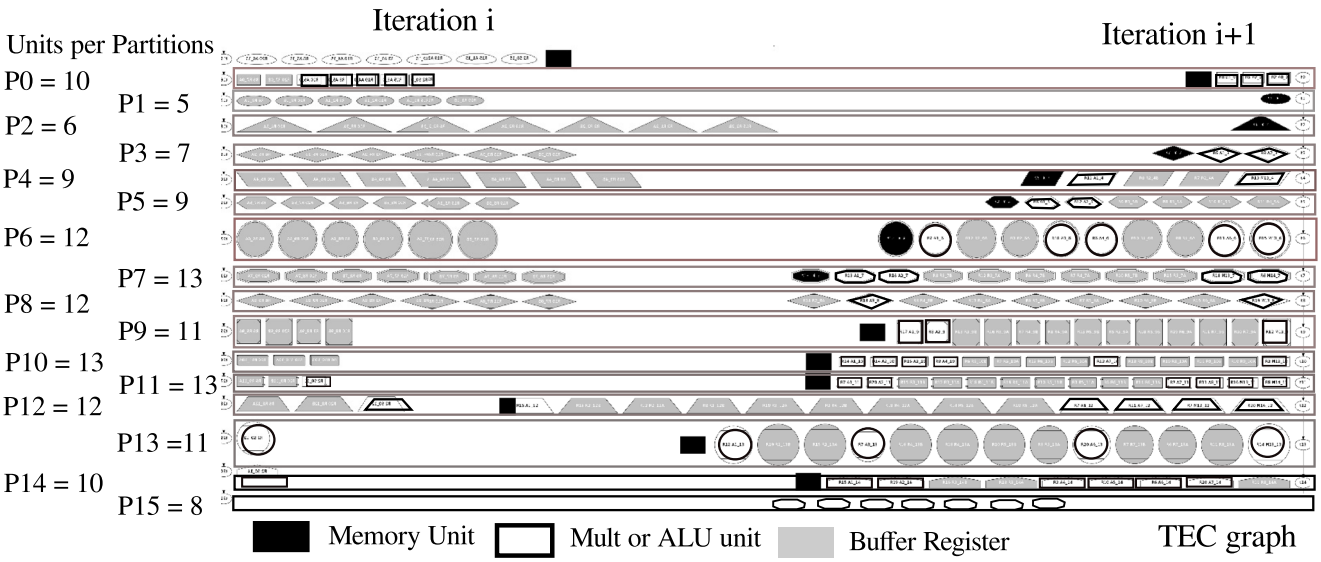


Figure 25 Cjpeg scheduling: The generated TEC graph for two overlap iterations.

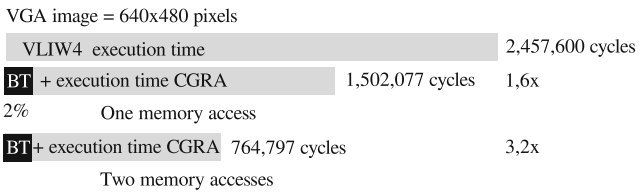


Figure 26 Execution Time for a VGA Image: 4-issue VLIW, BT MS + CGRA (One memory access per cycle and two memory accesses per cycle).

number of required units per partition is displayed. The black vertices represent the 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 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, the MS BT uses, in the worst case, 32 registers.

Assuming a VGA image with 640×480 pixels, the complete DCT application runs in 2,457,600 clock cycles. On the other hand, when including the accelerator and the binary translation overhead, the execution time reduces to 1,502,077 cycles as depicted in Fig. 26. Furthermore, a theoretical analysis also demonstrates a great potential to increase speedup. If we assume an aggressive scaling that enables an ILP of 8.88 by allowing 2 memory accesses per clock cycle, the execution time reduces to 764,797 cycles, which produces a speedup factor of 3.2x. The BT MS algorithm is implemented in C language, and it will be called during the execution as a trap function. For this example, the BT MS code executes in 27,517 cycles, which results in an overhead of 2% for the one memory access configuration and a VGA image. Additionally, for a 5 Mega-pixel image, which is a common size nowadays, the binary translation overhead is insignificant (0.06%).

Table 2 presents average number of clock cycles required to process one DFG node, in orders of magnitude, for nine modulo scheduling approaches found in literature: DRESC [27]; EMS [32]; RF [9]; RAM [29]; MSPR [11]; G -Minor [8]; EPImap [14]; REGIMap [16]; MS JIT [10]. Moreover, Table 2 presents our BT MS approach and a trace-based binary translation (TBT) proposed in [4]. The number of cycles were obtained from the respective references, with exception of DRESC, which time results were based on information reported in [8, 29]. The results show that with MS JIT [10] and our BT MS, a reduction ranging from two to six orders of magnitude was achieved. However, the MS JIT [10] time does not include the DFG extraction and binary translation. Moreover, the TBT [4] requires three orders of magnitude more efforts than our BT MS, which also includes a binary translation in software.

Finally, all 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 [28] 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 [35]. The functional units support the execution of all VLIW instructions.

Table 3 presents the results in amount of resources and maximum operation frequency after the placement and routing steps. The number of BRAM and LUT slices 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 VLIW-16 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 3. 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

Table 2 Compilation Time: Average Number of Clock Cycles required per DFG Node.

DRESC	RF, EMS REGIMap EPIMap Gminor	RAM	MSPR	MS JIT	TBT	BT MS
10^9	10^8	10^7	10^4	313	10^6	347

Table 3 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

of the both. Considering the 4-issue VLIW, the total area of our architecture is equivalent to a standalone 8-issue VLIW.

6 Related Work

The use of techniques to enhance performance through loop acceleration in CGRAs is not new. [28] was one of the first works to propose modulo scheduling for inner loop acceleration in CGRAs. To support loop exploitation, the authors also implemented a compiler that combines modulo scheduling, simulated annealing for placement and pathfinder for routing [27]. The solution performs mapping of instructions, represented as data dependency graphs (DDG), onto the architecture, represented in a modulo routing resource graph (MRRG). During compilation, the code passes through different transformations in order to generate the DDG for loops that can run in pipeline. In order to provide a high quality mapping, considering resources usage and routing, a cost function is computed and the simulated annealing is used to decide if the placement is acceptable or if this step should be performed again. According to the authors, the scheduling algorithm is time-consuming when compared to typical scheduling algorithms. The results demonstrate that minutes are necessary for scheduling instructions in a 64-FU reconfigurable architecture.

Other solutions [4, 8, 9, 14, 16, 29, 32] emerged in an attempt to reduce compile-time by improving the mapping algorithm. In [32], Park et al., proposed a modulo scheduling for CGRA, called EMS (Edge-centric Modulo Scheduling), that focuses on routing efficiency as the primary goal, since routing is a very time-consuming step in CGRAs. According to the authors, by investing in an efficient routing algorithm, it is possible to map dataflow graphs to the CGRA faster than the solutions that performs routing after scheduling. In order to find an efficient routing, the algorithm visits each individual edge and determines a routing cost using a routing cost function. The costs will indicate which is the best routing and consequently, the placement. The experimental results present a shorter compile-time in comparison to ADRES/DRESC solution around 1,185 seconds for EMS against 22,341 seconds for ADRES/DRESC. However, the mapping algorithm has a lower scheduling quality when compared to ADRES/DRESC, which results in performance reduction during execution time. [9, 29] also proposed solutions to reduce compile-time and sustaining quality of the scheduling algorithm. [9] proposes the use of placement and routing (P&R) code generation techniques as an alternative to register allocation algorithms and [29] makes use of recurrence cycle-aware scheduling technique, by grouping operations that belong to a recurrence cycle into a clustered node and computing a scheduling for those nodes. According to the authors, this solution presents better quality than simulated annealing solutions and reduction in compile-time.

EPIMap [14] is another approach targeted to improve mapping quality and reduce compile-time. According to the authors, the main contribution in their solution is the use of routing and re-computation to schedule data dependent instructions. One of the main problems when mapping instructions to CGRA occurs when the resource limitation causes data dependent instructions are scheduled in non-adjacent times. For instance, the first instruction is scheduledEPI in time t1 and the fourth instruction, which depends on the first one, is scheduled in time t3. In this case, the first instruction's result calculated in t1 must be stored until time t3, when it will be used. To solve this problem, [14] proposes the re-computation of some instructions, in this case, the same instruction is computed twice in different processing elements and the result of each unit is sent to a different instruction. This re-computation combined with a routing algorithm provides a higher quality scheduling than just using routing algorithm. Additionally, the EPIMap heuristic transforms an input data flow graph to an epimorphic equivalent graph that meets all the CGRA constraints. The algorithm performs a systematic search of the solution space, which ensures a higher quality mapping. The results presented in [14] demonstrate that, from 14 benchmarks, EPIMap achieved the best theoretical performance in 9 of them. Moreover, the performance improvement in comparison to EMS solution [32] is around 2.8x and compilation time is on average 30 seconds. The authors also present a formal model and NP-completeness proof for the modulo scheduling CGRA, demonstrating the complexity of the modulo scheduling algorithm for this type of architecture.

The mapping algorithm proposed in Chen and Mitra [8], called G-Minor, also performs graph transformations in the DFG's application in order to generate a high quality mapping in a reduced compile-time. According to the authors, the main advantage of the proposed approach in comparison to EPIMap [14] is the use of a customized graph minor testing procedure, which works only in subgraph mapping, consequently providing a faster mapping. The experimental results presented in [8] demonstrate the same scheduling quality when compared to EPIMap [14], with a faster compile-time of around 126 times faster. The average compile time for G-Minor approach is around 0.27 seconds.

In REGIMap approach [16], EPIMap's authors proposed a solution for a better usage of local register files by the mapping algorithm. The register files in each processing element are used to temporarily store data used in next cycles. This is a solution applied to solve the problem of data dependency among instructions scheduled in

different cycles. Through the use of theses registers, it is not necessary to hold the current value in the processing element. Experimental results comparing REGIMap and ADRES/DRESC [28] show performance increase of 1.89x and a reduction in compilation time of 56x. In spite of that, for a 4x4 mesh CGRA and varying number of local registers, the compilation time is still in order of thousands of seconds.

[4] proposes an approach that combines offline partitioning and mapping with online reconfiguration to accelerate loops in a reconfigurable coprocessor. The mapping algorithm searches for loop-based instruction traces, called megablocks. The detected megablocks are first transformed in a DFG representation and, then a translation mechanism transforms the detected instruction traces into a configuration. The application is originally described in MicroBlaze instruction traces and the reconfigurable processing unit is implemented in an FPGA. The experimental results include the analysis of 15 application kernels. The performance results indicate speedups of 1.26x up to 3.69x. The authors also evaluated the translation time for a fir filter megablock. The total time to translate a megablock implemented in assembly code to a configuration is on average 79 ms in 1 Ghz processor. According to the analysis, the most costly step (around 58%) is the conversion from assembly code to DFG representation. This result reassures how DFG extraction impacts on mapping time. The authors also mention the possibility to move partitioning and mapping to run-time, and presents preliminary results of a megablocks detector hardware. However, the consolidate run-time system is part of future works.

[37] proposes to partition the CGRA into clusters and schedule instructions from the same iteration into a single cluster. The authors state that, by using a cluster-based approach with a modulo scheduling algorithm, it is possible to reduce mapping time and increase the performance, since communication between distant processing elements is reduced. As in the previous solutions, the MS algorithm also works with a DFG. The results from three benchmarks demonstrate increase in performance when compared to G-Minor solution [8], around 9.8 %, as well as compilation time, around 6.5 %. However, compilation time is still in order of hundred of seconds.

All solutions mentioned above have in common the need for offline or compilation time solutions to analyze the application in order to find the kernels to run onto the CGRA. Additionally, the solutions also require the DFG extraction step. While a comparison among all solutions is not viable due to the lack of details in many works and the different benchmarks, it is possible to see that all of them require compilation time to perform instruction mapping, including scheduling and routing steps. From all the mentioned solutions, only the one proposed in [10] presents low compilation time and can be moved from compile-time to run-time. In spite of that, this solution still requires a compiler to perform the DFG extraction.

This work proposes a modulo scheduling that eliminates the DFG extraction and, through a greedy algorithm, performs a CGRA mapping faster than any of the solutions mentioned above. To the best of our knowledge, this work is the first one to introduce the run-time modulo scheduling algorithm from different ISA sources into a CGRA, ensuring software compatibility.

7 Conclusions

Coarse-grained reconfigurable architectures (CGRAs) have been widely adopted as a solution to accelerate application execution through instruction level parallelism exploitation. One of the approaches that has enabled significant performance enhancement is the combination of CGRA with modulo scheduling, which is a software pipeline technique that exploits parallelism among loop iterations. Application domains, such as signal processing and multimedia, directly benefit from these solutions, since they are composed of many software pipelining loops. In spite of the widespread use of modulo scheduling combined with CGRA found in literature, all proposed solutions work at compile-time. Even the most efficient solutions [10] require compile-time to perform part of the modulo scheduling algorithm. The main difficulties that preclude run-time MS solutions are caused by 1) mapping complexity, which includes placement, routing and scheduling, and it is a NP-complete problem. 2) Data dependence graphs (DDG) or dataflow graphs (DFG) extraction, which increases mapping time. 3) Use of mesh topology as interconnection, which also increases placement and routing complexity. In order to cope with the difficulties faced by the previous solutions, this paper proposed a novel binary translation mechanism as the first run-time modulo scheduling algorithm to map inner loops onto a coarse-grained reconfigurable architecture. The binary translation mechanism eliminates the DFG extraction by working directly with the assembly code. Another major advantage of working at the assembly level is software compatibility. To further reduce mapping time, the proposed modulo scheduling algorithm is a greedy approach, that finds the local optimal solution. Moreover, the proposed CGRA also uses a crossbar network [10], which reduces routing complexity. In this work, we also compared our approach to classical ILP architectures: superscalar Tomasulo, n-issue VLIW and a GPU. The comparison showed that, considering loop codes and the same amount of processing elements, the proposed approach reaches a

quasi-optimal ILP. The classical approaches with aggressive compiler techniques could achieve the performance of our BT MS approach, however our proposed algorithm executes on-the-fly with software compatibility.

In order to evaluate area, mapping quality and execution time of the MS algorithm, we presented a set of experiments comparing the proposed solution (a CGRA with a VLIW as host processor) with two other systems, the standalone VLIW processor (4-, 8- and 16-issue) and the REGIMap, which is currently the most efficient MS compiler-based approach. In spite of that, REGImap was not able to manage loop graphs with index counters with a large fanout. Regarding area occupancy, the proposed CGRA with 16 functional units (CGRA16) plus a 4-issue VLIW as host processor is equivalent to an 8-issue VLIW. Quality results were also evaluated and showed that the proposed run-time mechanism with CGRA16 achieved the quasi-optimal ILP. Finally, in performance evaluation, we presented an example of the proposed binary translation mechanism mapping a discrete cosine transformation loop to the CGRA with a speedup factor of 1.8 when compared to the same loop running onto a VLIW. In addition, to demonstrate the scalability of the proposed BT MS algorithm, we increased the amount of memory elements, from one memory access per clock cycle, to two memory accesses per clock cycle. In this case, the modulo scheduling was able to exploit this improvement and continue achieving quasi-optimal ILP. Future works include evaluating the acceleration considering larger application blocks and conditional branches [15], as well as on-the-fly generation of the CGRA using customized functional units.

References

1. Ahn, M., Yoon, J.W., Paek, Y., Kim, Y., Kiemb, M., Choi, K. (2006). A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proceedings DATE* (pp. 363–368).
2. Arnold, O., Matus, E., Noethen, B., Winter, M., Limberg, T., Fettweis, G. (2014). Tomahawk: Parallelism and heterogeneity in communications signal processing mpsocs. *ACM Transactions on Embedded Computing Systems*, *13*(3s), 107:1–107:241.
3. Beck, A.C.S., Rutzig, M.B., Gaydadjiev, G., Carro, L. (2008). Transparent reconfigurable acceleration for heterogeneous embedded applications. In *Proceedings of the conference on design, automation and test in Europe* (pp. 1208–1213).
4. Bispo, J., Paulino, N., Cardoso, J.M., Ferreira, J.C. (2013). Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units. *International Journal of Reconfigurable Computing*.
5. Bispo, J., Paulino, N., Ferreira, J., Cardoso, J. (2012). Transparent trace-based binary acceleration for reconfigurable hw/sw systems. *IEEE Transactions on Industrial Informatics*.
6. Boppu, S., Hannig, F., Teich, J. (2014). Compact code generation for tightly-coupled processor arrays. *Journal of Signal Processing Systems*, *77*(1–2), 5–29.
7. Bouwens, F., Berekovic, M., Kanstein, A., Gaydadjiev, G. (2007). Architectural exploration of the adres coarse-grained reconfigurable array. In *Proceedings ARC* (pp. 1–13).
8. Chen, L., & Mitra, T. (2012). Graph minor approach for application mapping on cgras. In *Proceedings FPT*.
9. De Sutter, B., Coene, P., Vander Aa, T., Mei, B. (2008). Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *Proceedings LCTES* (pp. 151–160).
10. Ferreira, R., Duarte, V., Meireles, W., Pereira, M., Carro, L., Wong, S. (2013). A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In *SAMOS XIII*.
11. Ferreira, R., Vendramini, J.G., Mucida, L., Pereira, M.M., Carro, L. (2011). An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proceedings CASES*.
12. Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., Hauck, S. (2009). Spr: an architecture-adaptive cgra mapping tool. In *Proceeding of the ACM/SIGDA international symposium on field programmable gate arrays, FPGA '09* (pp. 191–200). New York: ACM.
13. Goel, N., Kumar, A., Panda, P.R. (2014). Shared-port register file architecture for low-energy vliw processors. *ACM Transactions Architectural Code Optimization*, *11*(1).
14. Hamzeh, M., Shrivastava, A., Vrudhula, S. (2012). Epimap: Using epimorphism to map applications on CGRAs. In *Proceeding of DAC conference* (pp. 1280–1287).
15. Hamzeh, M., Shrivastava, A., Vrudhula, S. (2014). Branch-aware loop mapping on CGRAs. In *Proceeding of DAC conference on design automation conference* (pp. 1–6). ACM.
16. Hamzeh, M., Shrivastava, A., Vrudhula, S.B. (2013). Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *Proceeding of DAC conference* (p. 18).
17. Hartenstein, R. (2001). Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 asia and south pacific design automation conference, ASP-DAC '01*.
18. Hatanaka, A., & Bagherzadeh, N. (2007). A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In IPDPS 2007 (pp. 1–8).
19. Hoogerbrugge, J., & Corporaal, H. (1994). Register file port requirements of transport triggered architectures. In *Proceedings of the 27th annual international symposium on microarchitecture* (pp. 191–195). ACM.
20. Jääskeläinen, P., Kultala, H., Viitanen, T., Takala, J. (2014). Code density and energy efficiency of exposed datapath architectures. *Journal of Signal Processing Systems*, 1–16.
21. Kim, Y., Lee, J., Shrivastava, A., Yoon, J., Cho, D., Paek, Y. (2011). High throughput data mapping for coarse-grained reconfigurable architectures. *IEEE Transactions on CAD of International Circuits and Systems*, *30*(11), 1599–1609. doi:10.1109/TCAD.2011.2161217.
22. Laboratories, & H.P. (2014). Vex toolchain. http://www.hpl.hp.com/downloads/vex/.
23. Lee, G., Choi, K., Dutt, N. (2011). Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *30*(5), 637–650.
24. Lin, T.J., Chen, S.K., Kuo, Y.T., Liu, C.W., Hsiao, P.C. (2008). Design and implementation of a high-performance and complexity-effective vliw dsp for multimedia applications. *Journal of Signal Processing Systems*, *51*(3), 209–223.
25. Loeffler, C., Ligtenberg, A., Moschytz, G.S. (1989). Practical fast 1-d dct algorithms with 11 multiplications. *In 1989 international conference on acoustics, speech, and signal processing, 1989. ICASSP-89* (pp. 88–991). IEEE.

26. McCool, M. (2007). Signal processing and general-purpose computing and gpus [exploratory dsp]. *IEEE Signal Processing Magazine*, *24*(3), 109–114.
27. Mei, B., Vernalde, S., Verkest, D., De Man, H., Lauwereins, R. (2002). Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings FPT* (pp. 166–173).
28. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R. (2003). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings DATE*.
29. Oh, T., Egger, B., Park, H., Mahlke, S. (2009). Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings LCTES* (pp. 21–30).
30. Paek, J.K., Choi, K., Lee, J. (2011). Binary acceleration using coarse-grained reconfigurable architecture. *SIGARCH Computers Architecture News*, *38*(4), 33–39.
31. Park, H., Fan, K., Kudlur, M., Mahlke, S. (2006). Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proceedings CASES* (pp. 136–146).
32. Park, H., Fan, K., Mahlke, S.A., Oh, T., Kim, H., Kim, H.s. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings PACT*.
33. Park, H., Park, Y., Mahlke, S. (2009). Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings MICRO* (pp. 370–380).
34. Rau, B.R. (1994). Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings MICRO* (pp. 63–74).
35. Wong, S., Van As, T., Brown, G. (2008). ρ-vex: A reconfigurable and extensible softcore vliw processor. In *International conference on field-programmable technology FPT* (pp. 369–372). IEEE.
36. Yoon, J., Shrivastava, A., Park, S., Ahn, M., Jeyapaul, R., Paek, Y. (2008). Spkm: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proceedings ASPDAC* (pp. 776–782).
37. Zhou, L., Liu, H., Zhang, J. (2013). Loop acceleration by cluster-based cgra. *IEICE Electronics Express*, *10*(16).

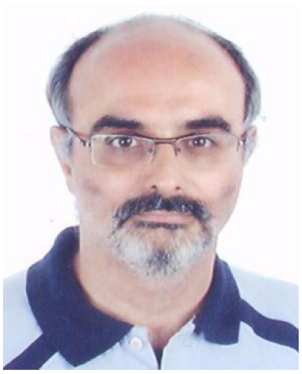
Ricardo Ferreira (SM'99) was born in Belo Horizonte, Brazil, in 1969. He received the B.E. degree in Physics and M.Sc. in Computer Science from the Federal University of Minas Gerais, Brazil, in 1991 and 1994, respectively, and the Ph.D degree in Applied Sciences (Microeletronics) from the Universite Catholique de Louvain, Louvain-la-Neuve, Belgium, in 1999. In 1992, he joined the Department of Computer Science, Federal University of Vicosa, as a Lecturer, and currently he is an Associate Professor. His current research interests include reconfigurable computing, FPGAs, GPUs, placement and routing, embedded systems, and run-time approaches.

Waldir Denver Muniz Meireles was born in Cel. Fabriciano, Brazil, in 1983. He received the degree in Computer Science from the Federal University of Vicosa in 2011 and master's degree in Computer Science from the Federal University of Vicosa in 2014 with emphasis on Reconfigurable Architectures. He is currently a Embedded Systems developer at iVision Systems Image and Vision SA. His research interests include Embedded Systems Design, Reconfigurable Hardware (CGRA) and FPGA, Sensor Network, unmanned vehicles and machine vision.

Monica Pereira received her Bachelor degree in Computer Science from Federal University of Rio Grande do Norte, Natal/Brazil in 2005. She received a M.Sc. in 2008 also in Computer Science from Federal University of Rio Grande do Norte and Ph.D. degree from Federal University of Rio Grande do Sul, Porto Alegre/Brazil in 2012. In 2010, she was at ITIV/Karlsruhe Institute of Technology, Karlsruhe/Germany, for an internship, where she developed activities related to fault tolerance in FPGA-based architectures. Since 2012, she is a Professor at Federal University of Rio Grande do Norte. Her research interests include reconfigurable architectures, embedded system design and fault tolerant architectures.

Stephan Wong was born in Paramaribo, Suriname on October 20th, 1973. He obtained his PhD from the Delft University of Technology in December 2002 after which I started as an assistant professor at the same university. His PhD thesis entitled "Microcoded Reconfigurable Embedded Processor" describes the MOLEN polymorphic processor, organization, and (micro-)architecture. His research interests include: Reconfigurable Computing, Distributed Collaborative Computing, High-Performance Computing, Embedded Systems, Hardware/Software Co-Design, Network Processing.

Carlos Arthur Lang Lisbôa received his degree in Civil Engineering in 1971, his Master in Computer Science Degree in 1976, and his PhD in Computer Science in 2008, all at Universidade Federal do Rio Grande do Sul, in Brazil. He is currently the head of the Applied Informatics Department of the Informatics Institute at the same university. His main fields of interest are computer architecture, hardware fault tolerance and embedded systems. He is member of the IEEE and of the Brazilian Computer Society (SBC).



Luigi Carro was born in Porto Alegre, Brazil, in 1962. He received the Electrical Engineering and the MSc degrees from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 1985 and 1989, respectively. From 1989 to 1991 he worked at ST-Microelectronics, Agrate, Italy, in the R&D group. In 1996 he received the Dr. degree in the area of Computer Science from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil. He is presently a full professor at the Applied Informatics Department at the Informatics Institute of UFRGS, in charge of Computer Architecture and Organization courses at the undergraduate levels. He is also a member of the Graduation Program in Computer Science at UFRGS, where he is co-responsible for courses on Embedded Systems, Digital signal Processing, and VLSI Design. His primary research interests include embedded systems design, validation, automation and test, fault tolerance for future technologies and rapid system prototyping. He has advised more than 20 graduate students, and has published more than 150 technical papers on those topics. He has authored the book Digital systems Design and Prototyping (2001-in Portuguese) and is the co-author of Fault- Tolerance Techniques for SRAM-based FPGAs (2006-Springer), Dynamic Reconfigurable Architectures and Transparent optimization Techniques (2010-Springer) and Adaptive Systems (Springer 2012). In 2007 he received the prize FAPERGS - Researcher of the year in Computer Science. His most updated resume is located in http://lattes.cnpq.br/8544491643812450. For the latest news, please check www.inf.ufrgs.br/~carro.