Chapter 2

# A Tightly Coupled VLIW/Reconfigurable Matrix and its Modulo Scheduling Technique

Bingfeng Mei,[1,2] Serge Vernalde,[2] Diederik Verkest,[1,2,3]
and Rudy Lauwereins[1,2]

[1] *IMEC vzw, Kalpeldreef 75, Leuven, Belgium*

[2] *Department of Electrical Engineering, Katholieke Universiteit Leuven, Belgium*

[3] *Department of Electrical Engineering, Vrije Universiteit Brussel, Belgium*

## 2.1 Introduction

Coarse-grained reconfigurable architectures have become increasingly important in recent years. Various architectures have been proposed [1–4]. These architectures often comprise a matrix of functional units (FUs), which are capable of executing word- or subword-level operations instead of bit-level ones found in common FPGAs. This *coarse* granularity greatly reduces the delay, area, power and configuration time compared with FPGAs. However, these advantages are achieved at the expense of flexibility. Usually the reconfigurable matrix alone is not able to execute entire applications. Most coarse-grained architectures are coupled with processors, typically RISCs. The computational-intensive kernels, typically loops, are mapped to the matrix, whereas the remaining code is executed by the processor. So far not much attention has been paid to the integration of these two parts. The coupling between the processor and the matrix is often loose, consisting essentially of two separate parts connected by a communication channel. This results in programming difficulty and communication overhead. In addition, the coarse-grained reconfigurable architecture consists of components which are similar to those used in processors. This resource-sharing opportunity is not extensively exploited in traditional coarse-grained architectures.

*P. Lysaght and W. Rosenstiel (eds.),*
*New Algorithms, Architectures and Applications for Reconfigurable Computing,* 15–28.

To address these problems, in this chapter we present an architecture called ADRES (*Architecture for Dynamically Reconfigurable Embedded System*), which tightly couples a VLIW (very long instruction word) processor and a coarse-grained reconfigurable matrix. The VLIW processor and the coarse-grained reconfigurable matrix are integrated into one single architecture but with two virtual functional views. This level of integration has many advantages compared with other coarse-grained architectures, including improved performance, a simplified programming model, reduced communication costs and substantial resource sharing.

Any new architecture can hardly be successful without good design methodology. Therefore, we developed a compiler for ADRES. The central technology is a novel modulo scheduling technique, which is able to map kernel loops to the reconfigurable matrix by solving placement, routing and scheduling simultaneously in a modulo constrained space. Combined with traditional ILP (instruction-level parallelism) compilation techniques, our compiler can map an entire application to the ADRES architecture efficiently and automatically.

This chapter is organized as follows: section 2.2 discusses the architectural aspects of ADRES; section 2.3 describes the modulo scheduling algorithm in details; section 2.4 presents the experimental results and section 2.5 concludes the chapter.

## 2.2 ADRES Architecture

### 2.2.1 Architecture Description

The ADRES architecture (Fig 2.1) consists of many basic components, including mainly FUs and register files(RFs), which are connected in a certain topology. The FUs are capable of executing word-level operations selected by a control signal. The RFs can store intermediate data. The whole ADRES matrix has two functional views, the VLIW processor and the reconfigurable matrix. These two functional views share some physical resources because their executions will never overlap with each other thanks to the processor/co-processor model. For the VLIW processor, several FUs are allocated and connected together through one multi-port register file, which is typical for VLIW architecture. Some of these FUs are connected to the memory hierarchy, depending on available ports. Thus the data access to the memory is done through the load/store operations available on those FUs.

For the reconfigurable matrix part, apart from the FUs and RF shared with the VLIW processor, there are a number of *reconfigurable cells* (RC) which basically comprise FUs and RFs too (Fig. 2.2). The FUs can be heterogeneous supporting different operation sets. To remove the control flow inside loops,

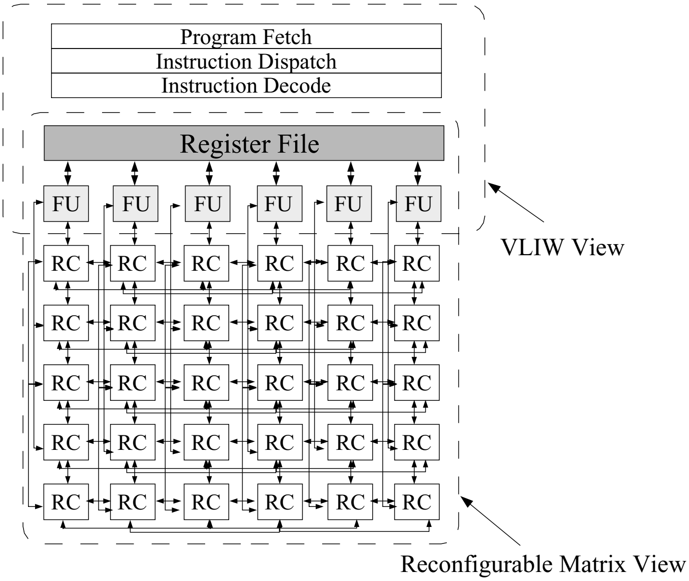

*Figure 2.1.* ADRES core

the FUs support predicated operations. The distributed RFs are small with less ports. The multiplexors are used to direct data from different sources. The configuration RAM stores a few configurations locally, which can be loaded on cycle-by-cycle basis. The configurations can also be loaded from the memory hierarchy at the cost of extra delay if the local configuration RAM is not big

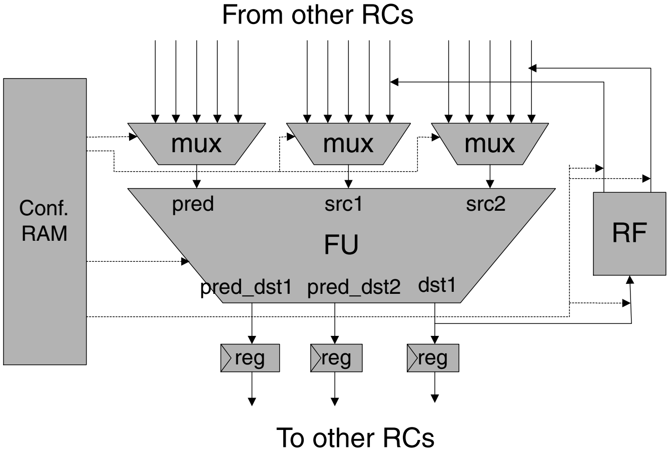

*Figure 2.2.* Example of a Reconfigurable Cell

enough. Like instructions in ISPs, the configurations control the behaviour of the basic components by selecting operations and multiplexors. The reconfigurable matrix is used to accelerate the dataflow-like kernels in a highly parallel way. The access to the memory of the matrix is also performed through the VLIW processor FUs.

In fact, ADRES is a template of architectures instead of a fixed architecture. An XML-based architecture description language is used to define the communication topology, supported operation set, resource allocation and timing of the target architecture [5]. Even the actual organization of the RC is not fixed. FUs and RFs can be put together in several ways. For example, two FUs can share one RF. The architecture shown in Fig. 2.1 and Fig. 2.2 is just one possible instance of the template. The specified architecture will be translated to an internal architecture representation to facilitate compilation techniques.

### 2.2.2 Improved Performance with the VLIW Processor

Many coarse-grained architectures consist of a reconfigurable matrix and a relatively slow RISC processor, e.g. TinyRisc in MorphoSys [1] and ARM in PACT XPP [3]. These RISC processors execute the unaccelerated part of the application, which only represents a small portion of execution time. However, such a system architecture has problems due to the huge performance gap between the RISC and the matrix. According to Amdahl's law [6], the performance gain that can be obtained by improving some portion of an application can be calculated according to equation 2.1. Suppose the kernels, representing 90% of execution time, are mapped to the reconfigurable matrix to obtain an acceleration of 30 times over the RISC processor, the overall speedup is merely 7.69. Obviously a high kernel speedup is not translated to a high overall speedup. Speeding up the unaccelerated part, which is often irregular and control-intensive code, is important for the overall performance. Though it is hard to exploit higher parallelism from it on the reconfigurable matrix, it is still possible to discover *instruction-level parallelism* (ILP) using a VLIW processor, where 2–4 times speedup over the RISC is reasonable. If we recalculate the speedup with the assumption of 3 times acceleration for the unaccelerated code, the overall acceleration is now 15.8, much better than the previous scenario. This simple calculation proves that using VLIW in ADRES can improve the overall speedup dramatically in certain circumstances.

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \tag{2.1}$$

### 2.2.3 Simplified Programming Model and Reduced Communication Cost

A simplified programming model and reduced communication cost are two important advantages of the ADRES architecture. These are achieved by making the VLIW processor and the reconfigurable matrix share access to the memory and the register file.

In other reconfigurable architectures, the processor and the matrix are essentially separated. The communication is often through explicit data copying. Though some techniques are adopted to reduce the data copying, e.g., wider data bus and DMA controller, the overhead is still considerable in terms of performance and energy. From the programming point of view, the separated processor and reconfigurable matrix require significant code rewriting. Starting from a software implementation, we have to identify the data structures used for communication and replace them with communication primitives. Data analysis should be done to make sure as few as possible data are actually copied. In addition, the kernels and the remaining code have to be cleanly separated in such a way that no shared access to any data structure remains. These transformations are often complex and error-prone.

In ADRES, the data communication is performed through the shared RF and memory. This feature is very helpful to map high-level language code such as C without major changes. When a high-level language is compiled to a processor, the local variables are allocated in the RF, while the static variables and arrays are allocated in the memory space. When the control of the program is transferred between the VLIW processor and the reconfigurable matrix, those variables used for communication can stay in the RF or the memory as they were. The copying is unnecessary because both the VLIW and the reconfigurable matrix share access to the RF and memory hierarchy. The code doesn't require any rewriting and can be handled by the compiler automatically.

### 2.2.4 Resource Sharing

Since the basic components such as the FUs and RFs of the reconfigurable matrix and those of the VLIW processor are basically the same, it is natural to think that resources might be shared to have substantial cost-saving. In other coarse-grained architectures, the resources cannot be effectively shared because the processor and the matrix are two separate parts. For example, the FU in the TinyRisc of MorphoSys cannot work cooperatively with the reconfigurable cells in the matrix. In ADRES, since the VLIW processor and the reconfigurable matrix are indeed two virtual functional views of the same physical entity, many resources are shared among these two parts. Due to its processor/co-processor model, only the VLIW processor or the reconfigurable matrix is active at any time. This fact makes resource sharing possible. Resource sharing of the powerful FUs and the multi-port RF of the VLIW by the matrix can greatly improve the performance and schedulability of kernels mapped on the matrix.

## 2.3 Modulo Scheduling

The objective of modulo scheduling is to engineer a schedule for one iteration of the loop such that this same schedule is repeated at regular intervals with respect to intra- and inter-iteration dependency and resource constraints. This interval is termed the *initiation interval* (II), essentially reflecting the performance of the scheduled loop. Various effective heuristics have been developed to solve this problem for both unified and clustered VLIWs [9, 11–13]. However, they cannot be applied to a coarse-grained reconfigurable architecture because the nature of the problem becomes more difficult, as illustrated next.

### 2.3.1 Problem Illustrated

To illustrate the problem, consider a simple dependency graph, representing a loop body, in Fig. 2.3a and a 2 × 2 matrix in Fig. 2.3b. The scheduled loop is depicted in Fig. 2.4a, where the 2 × 2 matrix is flattened to 1 × 4 for convenience of drawing. Nevertheless, the topology remains the same.

Fig 2.4a is a space-time representation of the scheduling space. From Fig. 2.4a, we see that modulo scheduling on coarse-grained architectures is a combination of 3 sub-problems: *placement, routing and scheduling*. Placement determines on which FU of a 2D matrix to place one operation. Scheduling, in its literal meaning, determines in which cycle to execute that operation. Routing connects the placed and scheduled operations according to their data dependencies. If we view *time* as an axis of 3D space, the modulo scheduling can be simplified to a placement and routing problem in a modulo-constrained 3D space, where the routing resources are asymmetric because any data can only

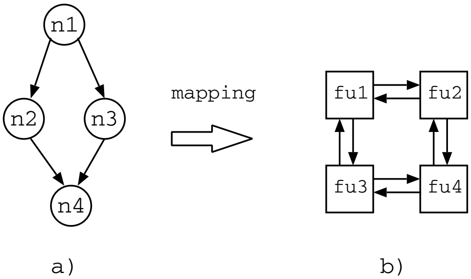

*Figure 2.3.* a) A simple dataflow graph; b) A 2 × 2 reconfigurable matrix

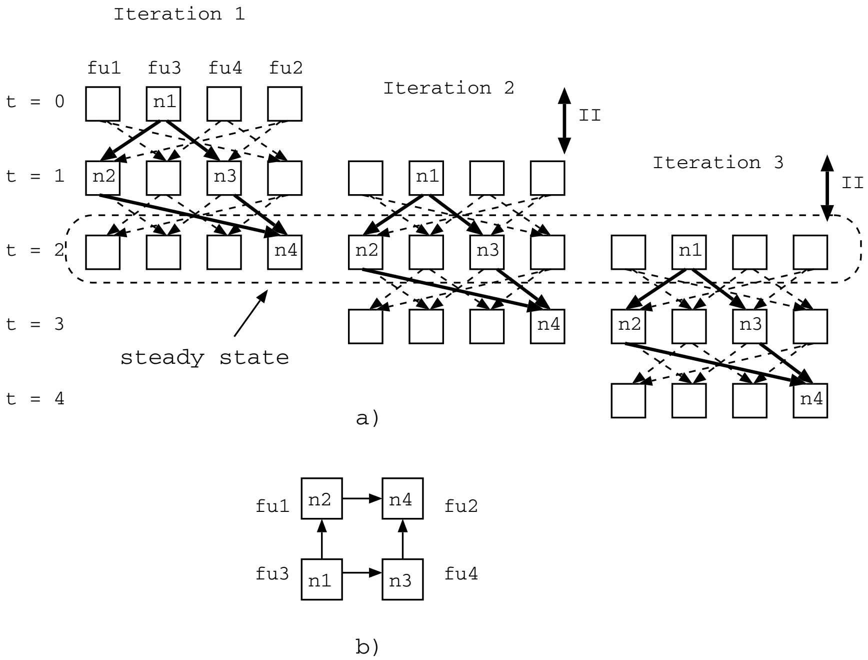

*Figure 2.4.* a) Modulo scheduling example; b) Configuration for 2 × 2 matrix

be routed from smaller time to bigger time, as shown in Fig. 2.4a. Moreover, all resources are modulo-constrained because the execution of consecutive iterations which are in distinct stages is overlapped. The number of stages in one iteration is termed *stage count* (SC). In this example, II = 1 and SC = 3. The schedule on the 2 × 2 matrix is shown in Fig. 2.4b. FU1 to FU4 are configured to execute n2, n4, n1 and n3 respectively. In this example, there is only one configuration. In general, the number of configurations that need to be loaded cyclically is equal to II.

By overlapping different iterations of a loop, we are able to exploit a higher degree of ILP. In this simple example, the instruction per cycle (IPC) is 4. As a comparison, it takes 3 cycles to execute one iteration in a non-pipelined schedule due to the data dependencies, corresponding to an IPC of 1.33, no matter how many FUs in the matrix.

## 2.3.2 Modulo Routing Resource Graph

We develop a graph representation, namely *modulo routing resource graph* (MRRG), to model the ADRES architecture internally for the modulo scheduling algorithm. MRRG combines features of the *modulo reservation table*(MRT) [7] for software pipelining and the *routing resource graph* [8] used in FPGA P&R, and only exposes the necessary information to the modulo scheduling algorithm. An MRRG is a directed graph $G = \{V, E, II\}$ which is constructed by composing sub-graphs representing the different resources of the ADRES architecture. Because the MRRG is a time-space representation of the architecture, every subgraph is replicated each cycle along the time axis. Hence each node v in the set of nodes V is a tuple $(r, t)$ where $r$ refers to the port of resource and $t$ refers to the time stamp. The edge set $E = \{(v_m, v_n) | t(v_m) <= t(v_n)\}$ corresponds to switches that connect these nodes. The restriction $t(v_m) <= t(v_n)$ models the asymmetric nature of the MRRG. Finally, II refers to the initiation interval. MRRG has two important properties. First, it is a modulo graph. If scheduling an operation involves the use of node $(r, t_j)$, then all the nodes $\{(r, t_k) | t_j \bmod II = t_k \bmod II\}$ are used too. Second, it is an asymmetric graph. It is impossible to find a route from node $v_i$ to $v_j$, where $t(v_i) > t(v_j)$. As we will see in section 2.3.3, this asymmetric property imposes big constraints on the scheduling algorithm. During scheduling we start with a minimal II and iteratively increase the II until we find a valid schedule (see section 2.3.3). The MRRG is constructed from the architecture specification and the II to try. Each component of the ADRES architecture is converted to a subgraph in MRRG.

Fig. 2.5 shows some examples. Fig. 2.5a is a 2D view of a MRRG subgraph corresponding to a FU, which means in the real MRRG graph with time dimension, all the subgraphs have to be replicated each cycle along the time

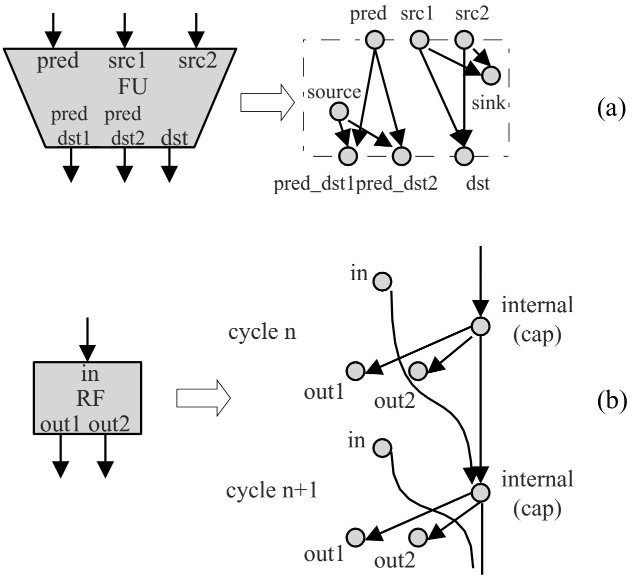

*Figure 2.5.* MRRG representation of ADRES architecture parts

axis. For FUs, all the input and output ports have corresponding nodes in the MRRG graph. Virtual edges are created between src1 and dst, src2 and dst, etc. to model the fact that a FU can be used as routing resource to directly connect src1 or src2 to dst, acting just like a multiplexor or demultiplexor. In addition, two types of artificial nodes are created, namely *source* and *sink*. When a commutative operation, e.g., *add*, is scheduled on this FU, the source or sink node are used as routing terminals instead of the nodes representing ports. Thus the router can freely choose which port to use. This technique improves the flexibility of the routing algorithm and leads to higher routability. Fig. 2.5b shows a space-time MRRG subgraph for a register file with one write port and two read ports. The idea is partly from [10]. Similar to the FU, the subgraph has nodes corresponding to each input and output port, which are replicated over each cycle. Additionally, an internal node is created to represent the capacity of the register file. All internal nodes along the time axis are connected one by one. The input nodes are connected to the internal node of next cycle, whereas the output nodes are connected to the internal node of this cycle. In this way, the routing capability of the register file is effectively modelled via its write-store-read functionality. Moreover, a *cap* property is associated with the internal node which is equal to the capacity of the register file. Therefore, the register allocation problem is implicitly solved by our scheduling algorithm. Other types of components such as bus and multiplexor can be modelled in a similar way. This abstraction, all routing resources, whether physical or virtual, are modelled in a universal way using nodes and edges. This unified abstract view of the architecture exposes only necessary information to the scheduler and greatly reduces the complexity of the scheduling algorithm.

### 2.3.3 Modulo Scheduling Algorithm

By using MRRG, the three sub-problems (placement, routing and scheduling) are reduced to two sub-problems (placement and routing), and modulo constraints are enforced automatically. However, it is still more complex than traditional FPGA P&R problems due to the modulo and asymmetric nature of the P&R space and scarcity of available routing resources. In FPGA P&R algorithms, we can comfortably run the placement algorithm first by minimizing a good cost function that measures the quality of placement. After minimal cost is reached, the routing algorithm connects placed nodes. The coupling between these two sub-problems is very loose. In our case, we can hardly separate placement and routing as two independent problems. It is very hard to find a placement algorithm and a cost function which can foresee the routability during the routing phase. Therefore, we propose a novel approach to solve these two sub-problems in one framework. The algorithm is described in Fig. 2.6.

```
SortOps();
II   := MII(DDG);

while not scheduled do
  InitMrrg(II);
  InitTemperature();
  InitPlaceAndRoute();       (1)

  while not scheduled do
    for each op in sorted operation list
      RipUpOp();

      for i := 1 to random_pos_to_try do
        pos := GenRandomPos();
        success := PlaceAndRouteOp(pos); (3)

        if success then
          new_cost := ComputeCost(op);   (4)
          accepted := EvaluateNewPos();  (5)
          if accepted then
            break;
          else
            continue;
        endif                                  (2)
      endfor

      if not accepted then
        RestoreOp();
      else
        CommitOp();

      if get a valid schedule then
        return scheduled;
    endfor

    if StopCriteria() then               (6)
      break;

    UpdateOverusePenalty();              (7)
    UpdateTemperature();                 (8)

  endwhile
  II++;
endwhile
```

*Figure 2.6.* Modulo scheduling algorithm for coarse-grained reconfigurable architecture

First all operations are ordered by the technique described in [11]. Priority is given to operations on the critical path and an operation is placed as close as possible to both its predecessors and successors, which effectively reduces the routing length between operations. Like other modulo scheduling algorithms, the outermost loop tries successively larger II, starting with an initial value equal to the minimal II (MII), until the loop has been scheduled. The MII is computed using the algorithm in [9].

For each II, our algorithm first generates an initial schedule which respects dependency constraints, but may overuse resources (1). For example, more than one operation may be scheduled on one FU in the same cycle. In the inner loop (2), the algorithm iteratively reduces resource overuse and tries to come up with a legal schedule. At every iteration, an operation is ripped up from the existing schedule, and is placed randomly (3). The connected nets are re-routed accordingly. Next, a cost function is computed to evaluate the new placement and routing (4). The cost is computed by accumulating the cost of all used MRRG nodes incurred by the new placement and routing of the operation. The cost function of each MRRG node is shown in eq. 2.2. It is constructed by taking into account the penalty of overused resources. In eq. 2.2, there is a basic cost (*base_cost*) associated with each MRRG node. The *occ* represents the occupancy of that node. The *cap* refers to the capacity of that node. Most MRRG nodes have a capacity of 1, whereas a few types of nodes such as the internal node of a register file have a capacity larger than one. The *penalty* factor associated with overused resources is increased at the end of each iteration (7). Through a higher and higher overuse penalty, the placer and router will try to find alternatives to avoid congestion. However, the penalty is increased gradually to avoid abrupt increases in the overused cost that may trap solutions into local minima. This idea is borrowed from the *Pathfinder* algorithm [8], which is used in FPGA P&R problems.

$$cost = base\_cost \times occ + (occ - cap) \times penalty \tag{2.2}$$

In order to escape from local minima, we use a *simulated annealing* strategy to decide whether each move is accepted or not (5). In this strategy, if the new cost is lower than the old one, the new P&R of this operation will be accepted. On the other hand, even if the new cost is higher, there is still a chance that the move may be accepted, depending on the "*temperature*". At the beginning, the temperature is very high so that almost every move is accepted. The temperature is decreased at the end of the each iteration (8). Therefore, the operation is increasingly difficult to move around. In the end, if the termination criteria is met without finding a valid schedule (6), the schedule algorithm starts with the next II.

## 2.4 Experimental Results

In the experiments, an architecture resembling the topology of MorphoSys [1] is instantiated from the ADRES template. In this configuration, a total of 64 FUs are divided into four tiles, each of which consists of 4 × 4 FUs. Each FU is not only connected to the 4 nearest neighbor FUs, but also to all FUs within the same row or column in this tile. In addition, there are row buses and column buses across the matrix. The first row of FUs is also used by the

*Table 2.1.* Scheduling results of kernels

| Loop | No. of ops | Live-in vars | Live-out vars | II | IPC | Sched. density |
|---|---|---|---|---|---|---|
| idct1 | 93 | 4 | 0 | 3 | 31 | 48.4% |
| idct2 | 168 | 4 | 0 | 4 | 42 | 65.6% |
| adpcm-d | 55 | 9 | 2 | 4 | 13.8 | 21.5% |
| mat_mul | 20 | 12 | 0 | 1 | 20 | 31.3% |
| fir_cpl | 23 | 9 | 0 | 1 | 23 | 35.9% |

VLIW processor, and are connected to a multi-port register file. Only the FUs in the first row are capable of executing memory operations, i.e., load/store operations.

The testbench consists of 4 programs, which are derived from C reference code of TI's DSP benchmarks and MediaBench [14]. The *idct* is a 8 × 8 inverse discrete cosine transformation, which consists two loops. The *adpcm-d* refers to an ADPCM decoder. The *mat_mul* computes matrix multiplication. The *fir_cpl* is a complex FIR filter. They are typical multimedia and digital signal processing applications with abundant inherent parallelism.

The schedule results are shown in Table 2.1. The second column refers to the total number of operations within the loop body. The II is *initiation interval*. The live-in and live-out variables are allocated in the VLIW register file. The instructions-per-cycle (IPC) reflects how many operations are executed in one cycle on average. Scheduling density is equal to *IPC/No. of FUs*. It reflects the actual utilization of all FUs for computation. The results show the IPC is pretty high, ranging from 13.8 to 42. The FU utilization is ranged from 21.5% to 65.6%. For kernels such as *adpcm-d*, the results are constrained by achievable minimal II (MII). The Table 2.2 shows comparisons with the VLIW processor. The tested VLIW processor has the same configuration as the first row of the tested ADRES architecture. The compilation and simulation results for VLIW architectures are obtained from IMPACT, where aggressive optimizations are enabled. The results for the ADRES architecture are obtained from a developed co-simulator, which is capable of simulating the mixed VLIW and reconfigurable matrix

*Table 2.2.* Comparisons with VLIW architecture

| App. | Total ops (ADRES) | Total cycles (ADRES) | Total ops (VLIW) | Total cycles (VLIW) | Speed-up |
|---|---|---|---|---|---|
| idct | 211676 | 6097 | 181853 | 38794 | 6.4 |
| adpcm_d | 8150329 | 594676 | 5760116 | 1895055 | 3.2 |
| mat_mul | 20010518 | 1001308 | 13876972 | 2811011 | 2.8 |
| fir_cpl | 69126 | 3010 | 91774 | 18111 | 6.0 |

code. Although these testbenches are small applications, the results already reflect the impact of integrating the VLIW processor and the reconfigurable matrix. The speed-up over the VLIW is from 2.8 to 6.4, showing pretty good performance.

## 2.5 Conclusions and Future Work

Coarse-grained reconfigurable architectures have been gaining importance recently. Here we have proposed a new architecture called ADRES, where a VLIW processor and a reconfigurable matrix are tightly coupled in a single architecture. This level of integration brings a lot of benefits, including increased performance, simplified programming model, reduced communication cost and substantial resource sharing. We also describe a novel modulo scheduling algorithm, which is the key technology in the ADRES compiler. The scheduling algorithm is capable of mapping a loop to the reconfigurable matrix to expolit high parallelism. The experiment results show great performance advantage over the VLIW processor with comparable design efforts.

However, we have not implemented the ADRES architecture at the circuit level yet. Therefore, many detailed design problems have not been taken into account and concrete figures such as area and power are not available. Hence, to implement the ADRES design is in the scope of our future work. On the other hand, we believe the compiler is even more important than the architecture. We will keep developing the compiler to refine the ADRES architecture from the compiler point of view.

## References

[1] H. Singh, et al. *MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications*. IEEE Trans. on Computers, 49(5):465–481, 2000

[2] C. Ebeling and D. Cronquist and P. Franklin. *RaPiD—Reconfigurable Pipelined Datapath*. Proc. of International Workshop on Field Programmable Logic and Applications (FPL), 1996

[3] PACT XPP Technologies. http://www.pactcorp.com

[4] T. Miyamori and K. Olukotun. *REMARC: Reconfigurable Multimedia Array Coprocessor*. International Symposium on Field Programmable Gate Arrays (FPGA), 1998

[5] B. Mei et al. *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures*. International Conference on Field Programmable Technology, 2002

[6] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996

[7] M. S. Lam. *Software pipelining: an effecive scheduling technique for VLIW machines*. Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, 1988

[8] C. Ebeling et al. *Placement and Routing Tools for the Triptych FPGA*. IEEE Trans. on VLSI, 3(12):473–482, 1995

[9] B. Ramakrishna Rau. *Iterative Modulo Scheduling*. Hewlett-Packard Lab: HPL-94–115, 1995

[10] S. Roos. *Scheduling for ReMove and other partially connected architectures*. Laboratory of Computer Enginnering, Delft University of Technology, 2001

[11] J. Llosa et al. *Lifetime-Sensitive Modulo Scheduling in a Production Environment*. IEEE Trans. on Computers, 50(3):234–249, 2001

[12] C. Akturan and M. F. Jacome. *CALiBeR: A Software Pipelining Algorithm for Clustered Embedded VLIW Processors*. Proc. ICCAD, 2001

[13] M. M. Fernandes et al. *Distributed Modulo Scheduling*. Proc. High Performance Computer Architecture (HPCA), 1999

[14] C. Lee et al. *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems*. International Symposium on Microarchitecture, 1997