Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling

B. Mei, S. Vernalde, D. Verkest, H. De Man and R. Lauwereins

Abstract: Coarse-grained reconfigurable architectures have become increasingly important in recent years. Automatic design or compilation tools are essential to their success. A modulo scheduling algorithm to exploit loop-level parallelism for coarse-grained reconfigurable architectures is presented. This algorithm is a key part of a dynamically reconfigurable embedded systems compiler (DRESC). It is capable of solving placement, scheduling and routing of operations simultaneously in a modulo-constrained 3D space and uses an abstract architecture representation to model a wide class of coarse-grained architectures. The experimental results show high performance and efficient resource utilisation on tested kernels.

1 Introduction

Coarse-grained reconfigurable architectures have become increasingly important in recent years. Various architectures have been proposed [1–4]. These architectures often consist of tens to hundreds 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, but at the expense of flexibility. Other features include predictable timing, a small configuration storage space, flexible topology, combination with a general-purpose processor etc. On the other hand, compared with a traditional 'coarse-grained' very long instruction word (VLIW), the partial connectivity of coarse-grained reconfigurable architectures makes them scalable but still cost- and power-efficient.

The target applications of these architectures, e.g. telecommunications and multimedia applications, often spend most of their time executing a few *time-critical code segments* with well-defined characteristics. So the performance of a whole application may be improved considerably by mapping these critical segments, typically loops, on a hardware accelerator. Moreover, these computation-intensive segments often exhibit a high degree of inherent parallelism. This makes it possible to use the abundant computation resources available in coarse-grained architectures.

Unfortunately, few automatic design and compilation tools have been developed to exploit the massive parallelism found in applications and extensive computation resources found in coarse-grained reconfigurable architectures. Some research [1, 4] uses structure- or GUI-based design tools to manually generate a design, which obviously limits the size of the design that can be handled. Some researchers [5, 6] focus on instruction-level parallelism (ILP) in limited scope, fail to make use of the coarse-grained architecture efficiently and in principle cannot reach higher parallelism than a VLIW. Some recent research has started to exploit loop-level parallelism (LLP) by applying pipelining techniques [7–10], but still suffers from severe limitations in terms of architecture or applicability (see Section 6).

To address these problems, this paper presents a modulo scheduling algorithm, which is a key part of our DRESC framework [11], to exploit LLP on coarse-grained architectures. Modulo scheduling is a software pipelining technique used in ILP processors such as VLIW to improve parallelism by executing different loop iterations in parallel [12]. Applied to coarse-grained architectures, modulo scheduling becomes more complex, being a combination of placement and routing (P&R) in a modulo-constrained 3D space. To the best of our knowledge, modulo scheduling has not been successfully applied to arbitrarily connected coarse-grained architectures. We propose an abstract architecture representation, modulo routing resource graph (MRRG), to enforce modulo constraints and describe the architecture. The algorithm combines ideas from FPGA P&R and modulo scheduling from VLIW compilation. The algorithm has been tested on a set of benchmarks, which are all derived from C reference code of TI's DSP benchmarks [13], and results show high performance and efficient resource utilisation on an 8×8 coarse-grained architecture.

2 Target architecture

Our target platforms are a family of coarse-grained reconfigurable architectures. As long as certain features are supported (see later), there is no hard constraint on the number of FUs and register files, and the interconnection topology of the matrix. This approach is similar to the work on KressArray [14]. The difference is that we integrate predicate support, distributed register files and configuration

IEE Proceedings online no. 20030833

doi: 10.1049/ip-cdt:20030833

Paper received 15th May 2003

The authors are with IMEC vzw, Kapeldreef 75, B-3001, Leuven, Belgium

B. Mei, D. Verkest, H. De Man and R. Lauwereins are also with the Department of Electrical Engineering, Katholic Universiteit Leuven, Leuven, Belgium

D. Verkest is also with the Department of Electrical Engineering, Vrije Universiteit Brussel, Brussel, Belgium

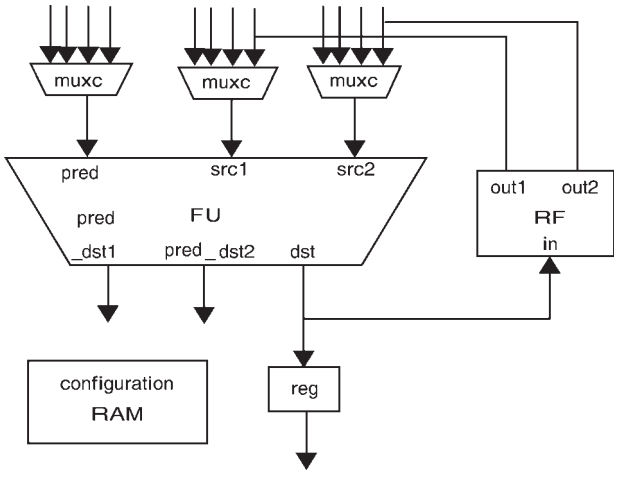

Fig. 1 *Example of FU and register file*

RAM to make the architecture template more generally applicable and efficient.

Basically, the target architecture is a regular array of functional units and register files. The FUs are capable of executing a number of operations, which can be heterogeneous among different FUs. To be applicable to different types of loops, the FU supports predicate operation. Hence, through if-conversion and hyperblock construction [15], while-loops and loops containing conditional statements are supported by the architectures. Moreover, predicate support is also essential in order to remove the loop-back operation and explicit prologue and epilogue. Register files (RF) provide small local storage space. The configuration RAM controls how the FU and multiplexers are configured, pretty much like instructions for processors. A few configurations are stored locally to allow rapid reconfiguration. Figure 1 depicts one example of organisation of FU and register file. Each FU has three input operands and three outputs. Each input operand can come from different sources, e.g. register file or bus, by using multiplexers. Similarly the output of a FU can be routed to various destinations such as inputs of neighbour FUs. It should be noted that the architecture template does not impose any constraint on the internal organisation of the FU and RF. Figure 1 is just one example of organisation of FU and RF. Other organisations are possible, e.g. two FUs sharing one register file.

At the top level, the FUs and RFs are connected through point-to-point connections or a shared bus for communication. Again, a very flexible topology is possible. Figure 2 shows two examples. In Fig. 2*a*, all neighbouring tiles have direct connections. In Fig. 2*b*, column and row buses are used to connect tiles within the same row and column. Using this template we can mimic many coarse-grained architectures found in the literature and also perform architecture exploration within the DRESC design space.

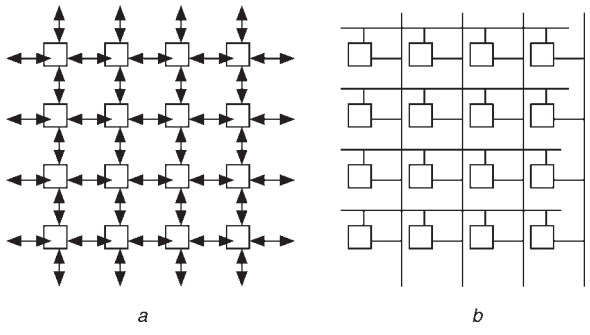

Fig. 2 *Examples of interconnection*

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 VLIW [16–19]. However, they cannot be applied to a coarse-grained reconfigurable architecture because the nature of the problem becomes more difficult, as illustrated next.

3.1 Problem illustrated

To illustrate the problem, consider a simple dependency graph, representing a loop body, in Fig. 3*a* and a 2×2 matrix in Fig. 3*b*. The scheduled loop is depicted in Fig. 4*a*, where the 2×2 matrix is flattened to 1×4 for convenience of drawing; however, the topology remains the same.

Figure 4*a* is a space-time representation of the scheduling space. From Fig. 4*a*, we see that modulo scheduling on coarse-grained architectures is a combination of three subproblems: *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 as a placement and routing problem in a modulo-constrained 3D space, where the routing resources are asymmetric because any data can only be routed from smaller time to bigger time, as shown in Fig. 4*a*. Moreover, all resources are modulo-constrained because the execution of consecutive iterations that 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. 4*b*. 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 three 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 are in the matrix.

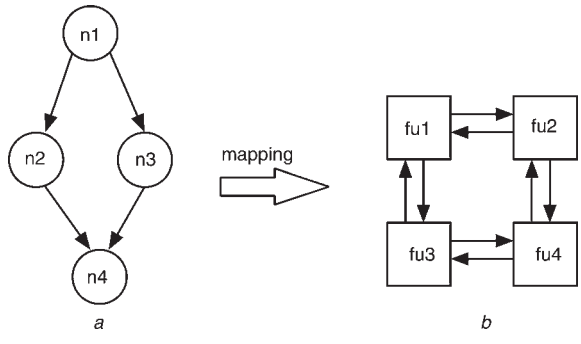

Fig. 3
a Simple dataflow graph
b 2×2 reconfigurable matrix

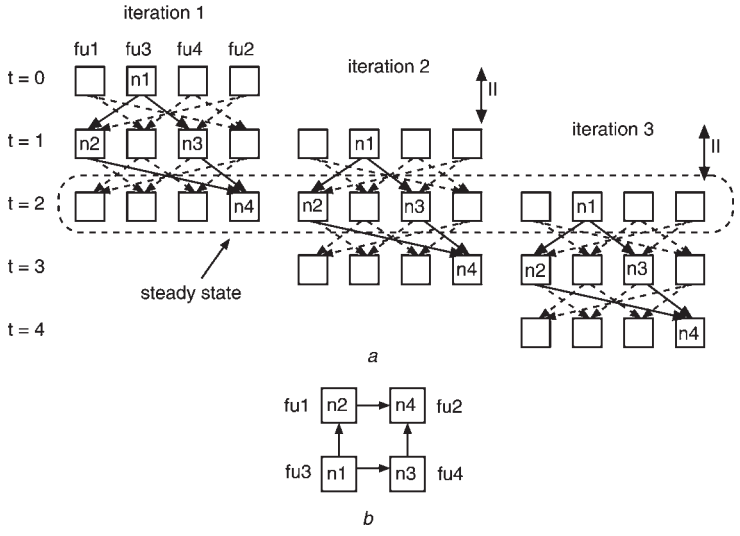

Fig. 4

a Modulo scheduling example
b Configuration for 2 × 2 matrix

3.2 Modulo routing resource graph

As shown in the previous Section, the modulo scheduling problem for coarse-grained architectures is essentially a P&R problem in a modulo-constrained 3D space. One difficult problem is how to model all heterogeneous routing resources in the 3D space. For example, FU may be used as routing resource. It can take data from one source port and copy it to the output port. RF can also serve as a routing resource. Data is written into an RF through one input port, and is read out later. This is essentially a routing capability along the time axis. Another problem is how to enforce the modulo constraint to make any modulo scheduling algorithm easier.

To address these problems, we propose a graph representation, namely *modulo routing resource graph* (MRRG), to model the architecture internally for the modulo scheduling algorithm. MRRG combines features of the *modulo reservation table* (MRT) [12] for software pipelining and the *routing resource graph* [20] used in FPGA P&R, and only exposes the necessary information to the modulo scheduling algorithm. A MRRG is a directed graph $G = \{V, E, II\}$, which is constructed by composing

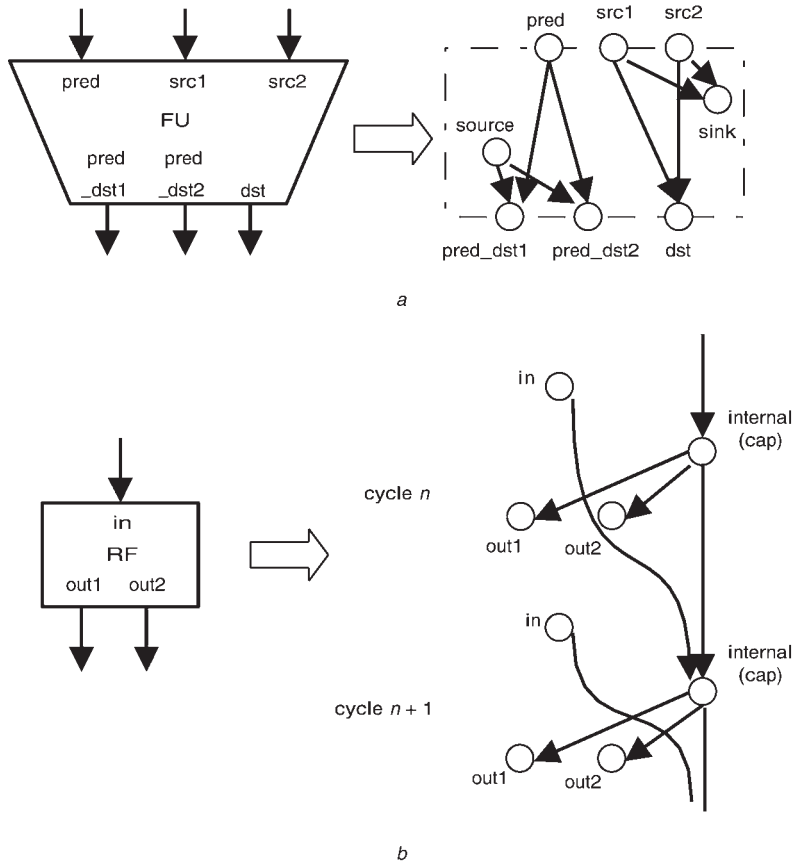

Fig. 5 *MRRG representation of DRESC architecture parts*

sub-graphs representing the different resources of the DRESC 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 $E = \{(v_m, v_n) \mid t(v_m) <= t(v_n)\}$ corresponds to switches that connect these nodes—the restriction $t(v_m) <= t(v_n)$ modelling 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), all the nodes $\{(r, t_k) \mid t_j \bmod II = t_k \bmod II\}$ 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 3.4, this asymmetric nature 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 3.4). The MRRG is constructed from the architecture specification and the II to try. Each component of the DRESC architecture is converted to a subgraph in MRRG.

Figure 5 shows some examples. Figure 5*a* is a 2D view of an MRRG subgraph corresponding to an FU, which means, in the real MRRG graph with time dimension, all the subgraphs have to be replicated each cycle along the time axis. For FU, 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 an FU can be used as routing resource to connect src1 or src2 directly to dst acting just like a multiplexer or demultiplexer. 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 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. 5*b* shows a space–time MRRG subgraph for an RF with one write port and two read ports. The idea is partly from [21]. 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 out of its write-store-read functionality. Moreover, a *cap* property is associated with the internal node, which is equal to the capacity of the RF. Therefore, the register allocation problem is implicitly solved by our scheduling algorithm (see Section 3.4).

Other types of components such as bus and multiplexer can be modelled in a similar way. By 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 only exposes necessary information to the scheduler and greatly reduces the complexity of the scheduling algorithm.

3.3 Prepare data dependency graph

The modulo scheduling algorithm takes a data dependency graph (DDG) representing the loop body and an MRRG representing the architecture as inputs. We use the IMPACT compiler framework [22, 23] as a frontend to parse C source code, do some optimisation and analysis, construct the required hyperblock [15] and emit the intermediate representation (IR), which is called *lcode*. Then various transformation and analysis passes are conducted to generate the DDG for detected pipelineable loops. Since the target reconfigurable architectures are different from traditional processors, we have developed some new techniques [11], e.g. a new method of removing prologue and epilogue code. Other transformations are borrowed from the VLIW compilation domain.

3.4 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 constraint is enforced automatically. However, it is still more complex than traditional FPGA P&R problem due to the modulo and asymmetric nature of the P&R space and scarce routing resources available. In FPGA P&R algorithms, we can comfortably run the placement algorithm first by minimising 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 almost impossible to find a placement algorithm and cost function that 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. 6.

First all operations are ordered by the technique described in [17]. 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 [16].

For each II, our algorithm first generates an initial schedule which respects dependency constraints, but may overuse resources ((1) in Fig. 6). 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 rerouted 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 the first equation. It is constructed by taking into account the penalty of overused resources. In the equation, 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). The second equation shows a simple scheme to update the penalty. Through a higher and higher overuse penalty, placer and router will try to find alternatives to avoid congestion. However, the penalty is increased gradually to avoid abrupt increase of the overused cost that may trap solutions into

```
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
```

Fig. 6 *Modulo scheduling algorithm for coarse-grained reconfigurable architecture*

local minima. This idea is borrowed from the *Pathfinder* algorithm [20], which is used in FPGA P&R problems:

$$cost = base\_cost \times occ + (occ - cap) \times penalty$$

$$penalty = penalty \times multi\_factor$$

In order to help solutions to escape from local minima, we uses a simulated annealing strategy to decide whether each move is accepted or not (5). In this stategy, 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 to accept the move, depending on *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. One key issue of simulated annealing is how the temperature is decreased. To achieve a good balance of quality and speed, we use an adaptive annealing technique [24] to update temperature:

$$T = \begin{cases} T \times 0.5, & accept\_rate \geq 0.96 \\ T \times 0.9, & 0.8 \leq accept\_rate < 0.96 \\ T \times 0.98, & 0.15 \leq accept\_rate < 0.8 \\ T \times 0.95, & accept\_rate < 0.15 \end{cases}$$

In this scheme, the accept rate is used to select a annealing rate among several ones (0.5 to 0.95), obtained from experiments. When the accept rate is in the middle range, the temperature is decreased slowly to ensure quality by exploiting an extensive search. When the accept rate is in the higher or lower range, the temperature decreases more rapidly to speed up the scheduling process.

In the end, if the stop criteria is met without finding a valid schedule (6), i.e., the scheduler can not reduce overused nodes after a number of iterations, the schedule algorithm starts with the next II.

According to [14], the maximum number of simultaneously live values at any cycle can approximate the number of registers required, which is called *MaxLive*. With the register file modeling discussed in section 3.2 and the scheduling algorithm described in this section, we can make sure the *MaxLive* is less than the capacity of the register file if we do find a valid schedule. Therefore, the register allocation problem is solved implicitly by the scheduler.

4 Codesign considerations

Usually pipelineable kernels only make up small portions of the application in terms of code size. The rest code is often control-intensive and executed by a processor. It is a kind of codesign problem in which an application is partitioned among the software and 'hardware', i.e. the reconfigurable matrix part. The communication and co-operation between these two parts are important issues. The DRESC framework is developed from the beginning with codesign in mind. We target a complete application instead of only pipelineable loops. When a high-level language is compiled to a processor, the local variables are normally allocated in the register file, whereas the static variables and arrays are allocated in the memory space. Some variables are accessed by both the pipelined kernels and the rest code, hence they serve as a communication channel between the software and 'hardware' parts. In the architecture, we assume that there is a register file that can be accessed by both the processor and the reconfigurable matrix.

During the preparation of the data dependency graph, the live-in and the live-out variables are identified. The live-in variables resemble input parameters, whereas live-out ones are like return values. Some variables can be both live-in and live-out. These variables have to be allocated in the shared register file so that they can be accessed by both the processor and the reconfigurable matrix. As described in previous Sections, the register file is modelled as a kind of routing resource. The scheduler can use it freely in the most favourable condition. This idea is in conflict with the fixed register allocation for live-in and live-out variables. Therefore, the modulo scheduling algorithm has to be adapted.

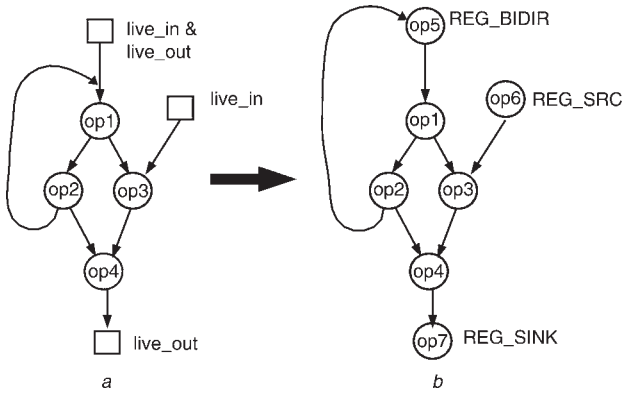

Fig. 7 *Transform live-in and live-out variables to pseudo operations*

a Original DDG
b Transformed DDG

In practice, we developed a technique to transform these variables to pseudo operations, namely REG_SOURCE, REG_SINK and REG_BIDIR operations. For example, in Fig. 7, the data dependency graph is transformed to a graph that only consists of operations and connecting edges. During the scheduling, these pseudo operations are treated and scheduled as normal operations with the extra constraint that they can only be assigned to and 'executed' by the shared register file. In this way, do not need to change the overall framework of the modulo scheduling algorithm. Those live-in and live-out variables are forced to stay in the shared register file, whereas the other variables may be allocated elsewhere, depending on how the scheduler routes the data dependency edge.

5 Experimental results

5.1 Experiment setup

We have tested our algorithm on an architecture that resembles the organisation or Morphosys [1]. In this configuration, a total of 64 FUs is divided into four tiles, each of which consists of 4×4 FUs. Each FU is connected to a local register file of size 8. Each FU is not only connected to the four nearest neighbour 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. All the FUs in the same row or column are connected to the corresponding bus. However, there are still significant differences with Morphosys. In Morphosys, the system consists of a general-purpose processor and a reconfigurable matrix. Our test architecture is a convergence of a VLIW processor and a reconfigurable matrix. The first row of FUs can work as a VLIW processor with the support of a multi-ported register file. Whenever the pipelined code is executed, the first row works co-operatively with the rest of matrix. For the other code, the first row acts like a normal VLIW processor, where instruction-level parallelism is exploited. The advantage of this convergence is twofold. First, since the FUs in a VLIW processor and reconfigurable matrix are similar, we can reuse many resources such as FUs and memory ports. Secondly, this convergence helps better integration of the reconfigurable matrix, which only accelerates certain kernels, and the rest of system. For example, live-in and live-out variables can be directly assigned to the VLIW register file, i.e. the one in the first row. The data copy cost between processor and matrix is therefore eliminated.

The testbench consists of four programs, which are all derived from the C reference code of TI's DSP benchmarks [13]. The *idct_ver* and *idct_hor* are vertical and horizontal loops of a is a 8×8 inverse discrete cosine transformation, where a simple optimisation is applied to transform nested loops to single-level ones. The *fft* refers to a radix-4 fast Fourier transformation. The *corr* computes 3×3 correlation. The *latanal* is a lattice analysis function. They are typical multimedia and digital processing applications with abundant inherent parallelism.

5.2 Scheduling results

The schedule results are shown in Table 1. The second column refers to the total number of operations within pipelined loops. The minimal initiation interval is the lower bound of achievable II, constrained by resources and recursive dependence, whereas the initiation interval is the value actually achieved during scheduling. The instructions per cycle (IPC) reflects how many operations are executed in one cycle on average. Scheduling density is equal to IPC/number of FUs. It reflects the actual utilisation of all FUs, excluding those used for routing. The last column is the CPU time to compute the schedule on a Pentium 4 1.7 GHz PC.

The IPC is high, ranging from 12 to 42. It is well above what can be obtained with any typical VLIW processor. For *idct_hor*, the IPC is especially high because its data dependency is mainly local. For *latanal*, the IPC is relatively low because it is constrained by MII. The CPU time to calculate the schedule is relatively long because of its SA-based search strategy and computational cost of each iteration.

5.3 Current limitations

Our scheduling algorithm has some limitations. First, it is relatively time-consuming compared with a typical scheduling algorithm of a compiler. Typically it takes minutes to schedule a loop of medium size. Secondly, at present it cannot handle some architecture constraints, e.g. pipelined FUs. Additionally, due to the way that the IMPACT front end constructs the hyperblock for a loop body [15], our scheduling algorithm can only handle the inner loop of a loop nest. This has an adverse impact on the overall performance of an application.

6 Related work

Several research projects have tried to apply pipelining techniques to reconfigurable architectures in order to obtain high performance. KressArray [7] uses simulated annealing to simultaneously solve the placement and routing sub-problems as well. However, it only handles the special case where II is equal to 1 because KressArray doesn't support multiple configurations for one loop. RaPiD [3] has a linear datapath that is a different approach compared with 2-dimensional meshes of processing elements. This restriction simplifies application mapping but restricts the design space dramatically. Similarly, Garp [8] also features a row-based architecture allowing direct implementation of a pipeline. It does not support multiplexing, so the implementation is inefficient when the II is bigger than 1. Recent work [9] tried to map loops directly to datapaths in a pipelined way. Lacking advanced scheduling techniques, it either uses

Table 1: Schedule results

Kernel	Number of operations	minimum initiation interval	initiation interval	instructions per cycle	Scheduling density	Time, s
idct_ver	93	2	3	31	44.8%	1446
idct_hor	168	3	4	42	65.6%	1728
fft	70	3	3	23.3	36.5%	1995
corr	56	1	2	28	43.8%	264
latanal	12	1	1	12	18.8%	6.5

a full-connected crossbar, or generates a dedicated datapath for several dataflow graphs, none of which is a good solution. PipeRench [10] uses a clever *pipeline reconfiguration* technique. The architecture is connected in a ring-like mode. Therefore, virtual pipeline stages can be mapped to physical pipeline stages in an efficient way. However, their technique is limited to very specific architectures, and thus cannot be applied to other coarse-grained reconfigurable architectures. Modulo scheduling algorithms on clustered VLIW architecture [18, 19] normally target a specific class of architectures and cannot handle arbitrarily connected architectures. In addition, the routing problem is virtually absent or rather easy to solve in clustered VLIW architectures.

7 Conclusions and future work

Coarse-grained reconfigurable architectures have advantages over traditional FPGAs in terms of delay, area and power consumption. In addition, they are more compiler-friendly because they possess features such as word- or subword-level operations and predictable timing. To exploit fully the potential of coarse-grained reconfigurable architectures, big problems to solve are: what kind of parallelism to exploit and how to extract it automatically.

We have developed a modulo scheduling algorithm to exploit loop-level parallelism on coarse-grained reconfigurable architectures, which resembles P&R algorithms for FPGAs. The results show up to 42 IPC and 65.6% FU utilisation for tested kernels, proving the potential for both coarse-grained reconfigurable architecture and our algorithm.

Overcoming the limitations of the modulo scheduling algorithm and better integration into the DRESC design flow will be our main focus in the future. For example, in order to handle nested loops, we have experimented with some source level transformations to replace a nested loop with a single loop. The preliminary results are very promising. The resource utilisation and parallelism for an IDCT kernel are improved by 3.64%, and the prologue and epilogue overhead is also reduced.

8 Acknowledgments

The authors would like to thank Prof. Henk Corporaal and Prof. Francky Cathoor for the insightful discussion about this work. This work is supported by a scholarship from the Katholieke Universiteit Leuven and IMEC, Belgium.

9 References

1 Singh, H., Lee, M.-H., Lu, G., Kurdahi, F.J., Bagherzadeh, N., and Chaves Filho, E.M.: 'Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications', *IEEE Trans. Comput.*, 2000, **49**, (5), pp. 465–481
2 Mirsky, E., and DeHon, A.: 'MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources'. Proc. IEEE Symp. on FPGAs for custom computing machines, Napa Valley, CA, 17–19 April 1996, pp. 157–166
3 Ebeling, C., Cronquist, D., and Franklin, P.: 'RaPiD—reconfigurable pipelined datapath'. Proc. Int. Workshop on Field programmable logic and applications, Darmstadt, Germany, 23–25 September 1996, pp. 126–135
4 *PACT XPP Technologies*, http://www.pactcorp.com, accessed 2003
5 Callahan, T.J., and Wawrzynek, J.: 'Instruction-level parallelism for reconfigurable computing'. Proc. Int. Workshop on Field programmable logic, Tallinn, Estonia, 31 August–1 September 1998, pp. 248–257
6 Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., and Amarasinghe, S.P.: 'Space-time scheduling of instruction-level parallelism on a RAW machine'. Proc. Architectural support for programming languages and operating systems (ASPLOS-VIII), San Jose, CA, 4–7 October 1998, pp. 46–57
7 Reiner Hartenstein and Rainer Kress, 'A datapath synthesis system for the reconfigurable datapath architecture', *Proc. ASP-DAC*, Mukukari, Japan, 29 August–1 September 1995, pp. 478–484
8 Callahan, T., and Wawrzynek, J.: 'Adapting software pipelining for reconfigurable computing'. Proc. Int. Conf. Compilers, architecture and synthesis for embedded systems (CASES), San Jose, CA, USA, 17–18 November 2000, pp. 57–64
9 Huang, Z., and Malik, S.: 'Exploiting operation level parallelism through dynamically reconfigurable datapath'. Proc. Design Automation Conference (DAC), New Orleans, LA, 2002, pp. 337–342
10 Schmit, H., Whelihan, D., Tsai, A., Moe, M., Levine, B., and Taylor, R.R.: 'PipeRench: a virtualized programmable datapath in 0.18 micron technology'. Proc. IEEE Custom Integrated Circuits Conference, Orlando, FL, 12–15 May 2002, pp. 63–66
11 Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R.: 'DRESC: a retargetable compiler for coarse-grained reconfigurable architectures'. Proc. Int. Conf. on Field Programmable Technology, Hong Kong, 16–18 December 2002, pp. 166–173
12 Lam, M.S.: 'Software pipelining: an effective scheduling technique for VLIW machines'. Proc. ACM SIGPLAN Conference on Programming language design and implementation, Atlanta, GA, 22–24 June 1988, pp. 318–327
13 TI Inc., 2002, http://www.ti.com/, accessed 2002
14 Hartenstein, R., Hertz, M., Hoffmann, Th., and Nageldinger, U.: 'KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array architectures'. Proc. ASP-Design Automation Conference, Yokohama, Japan, 25–28 January 2000, pp. 163–168
15 Mahike, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., and Bringmann, R.A.: 'Effective compiler support for predicated execution using the hyperblock'. Proc. 25th Annual Int. Symp. on Microarchitecture, Portland, OR, 1–4 December 1992, pp. 45–54
16 Ramakrishna Rau, B.: 'Iterative modulo scheduling'. Hawlett-Packard Lab: HPL-94-115, 1995
17 Llosa, J., Ayguade, E., Gonzalez, A., Valero, M., and Eckhardt, J.: 'Lifetime-sensitive modulo scheduling in a production environment', *IEEE Trans. Comput.*, 2001, **50**, (3), pp. 234–249
18 Akturan, C., and Jacome, M.F.: 'CALiBeR: a software pipelining algorithm for clustered embedded VLIW processor'. Proc. Int. Conf. on Computer Aided Design, San Jose, CA, 4–8 November 2001, pp. 112–118
19 Fernandes, M.M., Llosa, J., and Topham, N.P.: 'Distributed modulo scheduling'. HPCA, Orlando, FL, 9–12 January 1999, pp. 130–134
20 Ebeling, C., McMurchie, L., Hauck, S., and Burns, S.: 'Placement and routing tools for the Triptych FPGA', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1995, **3**, pp. 473–482
21 Roos, S.: 'Scheduling for ReMove and other partially connected architectures'. Laboratory of Computer Engineering, Delft University of Technology, Netherlands, 2001
22 'The IMPACT group'. http://www.crhc.uiuc.edu/impact, accessed 2002
23 Chang, P.P., Mahike, S.A., Chen, W.Y., Warter, N.J., and Hwu, W.W.: 'IMPACT: an architectural framework for multiple-instruction-issue processors'. Proc. 18th Int. Symp. Computer Architecture (ISCA), Toronto, Canada, 27–30 May 1991, pp. 266–275
24 Betz, V., Rose, J., and Marguardt, A.: 'Architecture and CAD for deep submicron FPGAs' (Kluwer Academic Publishers, Boston, MA, 1999)