DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures

Bingfeng Mei[†‡], Serge Vernalde[†], Diederik Verkest[†*], Hugo De Man[†‡], Rudy Lauwereins[†‡]
[†]IMEC vzw, Kapeldreef 75, B-3001, Leuven, Belgium
[‡] Department of Electrical Engineering, Katholic Universiteit Leuven, Leuven, Belgium
[*] Department of Electrical Engineering, Vrije Universiteit Brussel, Brussel, Belgium

Abstract – *Coarse-grained reconfigurable architectures have become increasingly important in recent years. Automatic design or compiling tools are essential to their success. In this paper, we present a retargetable compiler for a family of coarse-grained reconfigurable architectures. Several key issues are addressed. Program analysis and transformation prepare dataflow for scheduling. Architecture abstraction generates an internal graph representation from a concrete architecture description. A modulo scheduling algorithm is key to exploit parallelism and achieve high performance. The experimental results show up to 28.7 instructions per cycle (IPC) over tested kernels.*

1 Introduction

Coarse-grained reconfigurable architecture have become increasingly important in recent years. Various architectures are proposed [1, 2, 3, 4, 5]. 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, however, at the expense of flexibility. Other features include predictable timing, a small instruction storage space, flexible topology, combination with a general-purpose processor, etc. Armed with much more computation resources than normal programmable devices such as RISC and VLIW processors, they promise to deliver higher performance or better performance/energy efficiency.

The target applications of these architectures, e.g., digital communications and multimedia consumer electronics, often spend most of their time executing a few *time-critical code segments* with well-defined characteristics. Therefore, the performance of whole application can be improved considerably by mapping these critical code segment on a hardware accelerator. Moreover, these computation-intensive segments often exhibit a high degree of inherent parallelism, i.e., a lot of operations can be executed concurrently. This fact makes it possible to make use of the abundant computation resources available in coarse-grained architectures.

Unfortunately, few automatic design and compiling tool has been developed to exploit the massive parallelism found in applications and extensive computation resources of coarse-grained reconfigurable architectures. Some research [1, 4] uses structure- or GUI-based design tools to manually generate design, which would have difficulty to handle big designs. Some researchers [6, 7] only focus on instruction-level parallelism (ILP), fail to make use of the coarse-grained architecture efficiently and in principle can't result in higher parallelism than a VLIW; Some recent research [8, 9] starts to exploit loop-level parallelism (LLP) by applying pipelining techniques, but can only handle small kernels due to lacking support multiplexing in architecture or scheduling algorithm.

In this paper, we present a retargetable compiler called DRESC (Dynamically Reconfigurable Embedded System Compiler). It is able to parse, analyze, transform, and schedule plain C source code to a family of compiler-friendly coarse-grained reconfigurable architectures, which have great flexibility in terms of amount of computation and storage resources, and interconnection topology. DRESC is focused on exploiting loop-level parallelism on a wide range of loops. The experimental results show up to 28.7 instructions per cycle (IPC) can be achieved over tested loops.

For those who are not familiar with terminology of VLIW compilation, they are referred to [10].

2 The Target Architecture

Since we aim to develop a retargetable compiler, our target platforms are actually a family of coarse-grained reconfigurable architectures. As long as certain features are supported (see further), there is no hard constraint on the amount of FUs, the amount of register files, and the interconnection topology of the matrix. This ap-

0-7803-7574-2/02/$17.00 ©2002 IEEE

proach is similar to the work of KressArray [11]. The difference is that we integrate predicate support, distributed register files and configuration RAM to make the architecture template more generally applicable and efficient.

Basically, DRESC 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. Each FU has a small configuration RAM, thus multiple configurations can be stored locally. To be applicable to more different types of loops, the FU supports predicate operation. Hence, through if-conversion and hyperblock construction [12], while-loops and loop containing conditional statements are supported by the DRESC architectures. Moreover, predicate-guarded operation is also essential in order to remove feedback operation, prologue and epilogue. Register files provide small local storage space and act as a kind of routing resource, as shown at section 5. Fig. 1 depicts one example of organization of FU and register file. Each FU has 3 input operands and 3 outputs. Each input operand can come from different sources, e.g. register file or bus, by using multiplexer. Similarly, the output of FU can be routed to various destinations such as input of neighbour FUs. The configuration RAM provides information to control how the FU and multiplexers are configured, pretty much like instructions for processors. It should be noted that DRESC architecture doesn't impose any constraint on the internal organization of the FU and RF. Fig. 1 is just one example of organization of FU and RF. Other organizations are possible, for example, two FUs sharing one register file.

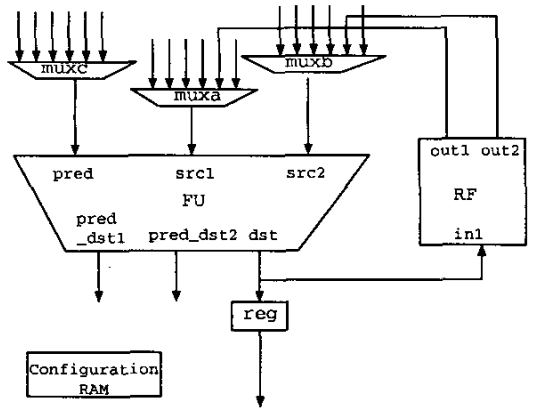

Figure 1: Example of FU and register file

At the top level, the FUs and register files are connected through point-to-point connections or a shared bus for communication. Again, a very flexible topology is possible. Fig. 2 shows two examples. In fig. 2a, all neighbour tiles have direct connections. In fig. 2b, column and row bus are used to connect tiles within same row and column. This flexibility allows to do architecture exploration to find the optimal one within DRESC design space. However, this architecture exploration is beyond the scope of this paper.

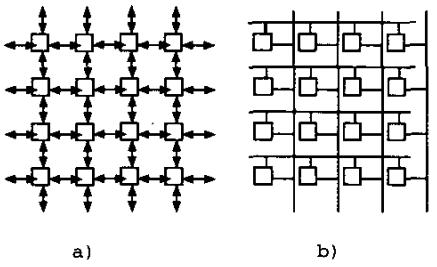

Figure 2: Examples of interconnection and topology

3 The Structure of DRESC Compiler

Fig. 3 shows the overall structure of our compiler. We heavily rely on IMPACT compiler framework[13] as a frontend to parse C source code, do some optimization and analysis, construct the required hyperblock, and emit the intermediate representation (IR), which is called *lcode*. Moreover, IMPACT is also used as a library in DRESC implementation to parse *lcode*, on the basis of which DRESC's own internal representation is constructed.

Taking *lcode* as input, various analysis passes are executed to obtain necessary information for later transformation and scheduling, for instance, pipelinable loops are identified and predicate-sensitive dataflow analysis is performed to construct a data dependency graph (DDG). Next, a number of program transformations are performed to build a scheduling-ready pure dataflow used by the scheduling phase. Since the target reconfigurable architectures are different from traditional processors, some new techniques are developed, while others are mostly borrowed from VLIW compilation domain. More details on the analysis and transformation techniques can be found in section 4.

In the right-hand side of fig. 3, the architecture description and abstraction path is shown. Our goal is to use an XML-based language to describe the various aspects of the target architecture. An architecture parser , which is still under development, translates the description to an internal architecture description format. From this internal format, an architecture abstrac-

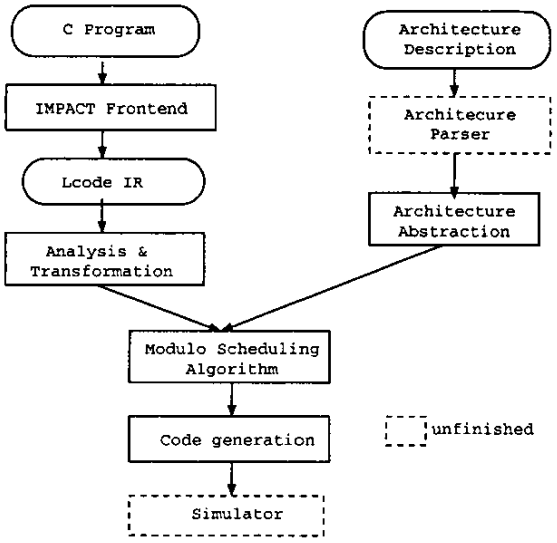

Figure 3: The structure of DRESC compiler

tion step generates a modulo routing resource graph (MRRG) which is used by the modulo scheduling algorithm. Details of the architecture abstraction step are discussed in section 5.

The modulo scheduling algorithm plays a central role in the DRESC compiler because we believe a major strength of coarse-grained reconfigurable architectures is in loop-level parallelism. At this point, both program and architecture are represented in the forms of graphs. The task of modulo scheduling is to map the program graph to the architecture graph and try to achieve optimal performance while respecting all dependencies. Section 6 illustrates the algorithm in depth. After that, the scheduled code is fed to a simulator, which we expect not only is able to measure the performance but also the power consumption. This simulator is still under development.

Since the DRESC compiler is only focused on exploiting parallelism for coarse-grained reconfigurable architectures, we don't pay much attention to other common issues of developing a compiler, e.g., the scheduling problem for the remaining code (non-loop kernel). We assume these problems are well solved for instruction-set processors and beyond the scope of this paper.

4 Program Analysis and Transformation

The purpose of program analysis and transformation in the DRESC compiler is to prepare correct and efficient dataflow graphs for loop pipelining. Most steps are described below.

Identifying Pipelinable Loops. Each hyperblock or basic block is checked to see if it is a loop candidate. The DRESC compiler can handle both FOR loops and WHILE loops. However, not all loops can be pipelined. The loops with function calls, break or continue statements are excluded. Moreover, only innermost loops can be pipelined.

Data Dependence Graph Construction. Precise DDG is very important for the DRESC compiler. The weak dependence analysis yields over-conservative DDG and leads to adverse impacts on performance. In DRESC architecture, predicate operation is supported. Therefore, we adopt the analysis algorithm based on *binary decision diagrams* (BDD) [14]. In constructed DDG, there are two types of edges: data edges and precedence edges. Data edges indicate that data need to be routed between the connected operation terminals. Precedence edges indicate that the operations need to be ordered, but no data is routed between the operations.

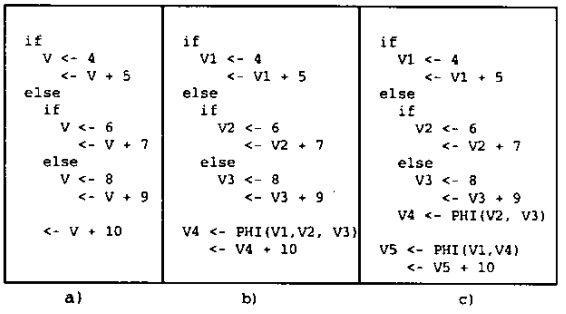

Figure 4: a) Original code b) SSA form c) Normalized SSA form

Normalized Static Single Assignment Form (SSA). SSA form is normally used as an intermediate format to remove false dependencies, i.e., the output and anti-dependencies. Fig. 4 shows how a piece of code is transformed to SSA form. Each assignment to a variable is given a unique name. At join node, a special form of assignment called ϕ-function is added. However, the SSA form is not intended for implementation due to the created ϕ-function. Therefore, we proposed a normalized SSA form tailored to coarse-grained architectures for direct implementation. Each ϕ-function now can only take two inputs. The selection is controlled by a signal produced by predicate analysis. Essentially it works like a multiplexer. When mapped to FU, two input are mapped to *src1* and *src2* operands, and the control signal is mapped to *pred* operand. A special PHI operation is added to FU's instruction set.

Live-in and Live-out Analysis. Live-in and live-out

variables serve as the communication channel between the pipelined loop and the rest of the application. Unlike other variables in the loop, they are often assigned to a fixed register file. Accurate live-in and live-out analysis helps to reduce communication load on that register file. We detect these variables by conducting liveness analysis in the loop body and its descendants at control data flow graph(CDFG) representing entire application.

Removing Explicit Prologue and Epilogue. Pipelining always introduces prologue and epilogue. The prologue is defined as from the start of the loop to the time that the first iteration finishes. At that point, the loop reaches a steady state where all pipeline stages are filled. Similarly the epilogue can be defined as from the start of the last iteration to the time the loop completes. Suppose there are n stages in the pipeline, the length of both prologue and epilogue are $n-1$. There are several schemes to generate prologue and epilogue codes for VLIWs. One category emits the prologue and epilogue as separate code. The other solution uses the same code for prologue, epilogue and kernel but with special hardware support to control which parts of the code are actually executed. The first solution is very costly for coarse-grained reconfigurable architectures. The kernel itself only requires II (initiation interval) configurations. The prologue and epilogue would require $2 \times (n-1) \times II$ configurations. To load and store these extra configurations is expensive in terms of configuration time and storage space. Therefore, we developed a technique to remove explicit prologue and epilogue. At the same time, the feedback operation of the loop is also removed to form a pure dataflow graph ready for scheduling. Fig. 5 illustrates how they are removed.

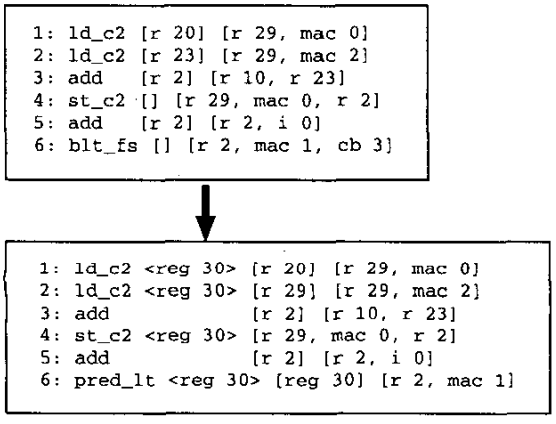

Figure 5: Remove prologue, epilogue and feedback operation

First the branch operation 6 is replaced with a operation that generates a predicate operand by input conditions. Next all *unsafe* operations are marked. Unsafe operations are defined as those that might damage data integrity or affect the performance during prologue and epilogue, e.g., the load/store operations and the operations modify live-out variables. Finally, a new predicate operand, reg 30 in fig. 5, is inserted to protect each unsafe operation. Operation 6 itself also has a predicate operand, reg 30. In the scheduling phase, the timing requirement to control each pipeline stage will be naturally met. This technique works like the sequencer in [8], but doesn't require specific hardware support.

Calculation of Minimum Initiation Interval (MII). The minimum initiation interval, MII, is the larger of the resource-constrained MII, ResMII, and the recurrence-constrained MII, RecMII. They are computed using the algorithm from [15]. At the same time, ASAP (as soon as possible), ALAP (as late as possible), and *mobility* are computed. They are used in scheduling and operation ordering.

Operation Ordering. The ordering phase takes the dependence graph as input and produces an ordered list of operations, indicating the order in which the operations will be scheduled. The priority is granted to operations with more constraints, e.g., the operations located on the critical path. At the same time, the operation should be placed as close as possible to both its predecessors and successors in order to reduce routing distance. Therefore, we use the algorithm from [16] for this task.

5 Architecture Abstraction

After parsing architecture description, the internal architecture representation says how many FUs and register files the architecture has and how they are connected. This description is complete but not appropriate for use by the scheduling algorithm. Moreover, since we are focused on modulo scheduling algorithm, an abstract representation like *modulo reservation table* (MRT) [17] for software pipelining is required to enforce modulo constraints. As pointed out in section 6, the modulo scheduling problem for coarse-grained architectures is indeed a placement and routing (P&R) problem with modulo constraints and asymmetric routing resources in a 3-dimensional (3D) space. Therefore, 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 MRT for software pipelining and the routing resource graph [18] used in FPGA P&R, and only exposes the necessary information to the modulo scheduling algorithm. A MRRG is

a directed graph $G = \{V, E, II\}$. The set of nodes V corresponds to the ports or wires or artificially created nodes (see further) in DRESC architecture. Each node is associated with a time t. The edge set $E = \{(v_i, v_j); t_{v_i} \leq t_{v_j}\}$ corresponds to switches that connect these nodes. II is a modulo time referring to *initiation interval*. MRRG has two important properties. First, it is a modulo graph. If scheduling an operation involves the use of node R at time T, then all the nodes with same [(T mod II), R] value 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 6, this asymmetric nature imposes big constraints on the scheduling algorithm. Therefore, we can't view placement and routing as two separate problems as in FPGA's P&R algorithms.

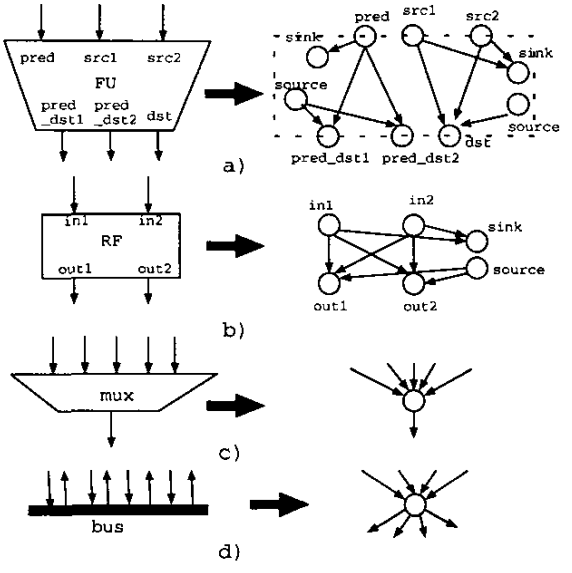

Figure 6: MRRG representation of DRESC architecture parts

MRRG is constructed from the architecture specification and the II to try (see the section 6). Each component of DRESC architecture is converted to a subgraph in MRRG. Fig. 6 shows some examples. Fig. 6a is a 2D view of a MRRG subgraph corresponding to a FU, which means in 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 FU can be used as routing resource to directly connect src1 or src2 to dst, acting just like a multiplexer or demultiplex or. In addition, two types of artificial nodes are created, namely *source* and *sink*, to model the logically equivalent ports. When a operation is scheduled on this FU, the source and sink nodes are occupied instead of nodes of ports meaning the router can freely choose which port to use as long as it is logically equivalent with the one specified by the operation. This technique improves the flexibility of routing algorithm, and eventually leads to higher routability. Fig. 6b shows a MRRG subgraph for a register file. Similar to FU, there are also artificially created *source* and *sink* nodes, which are introduced to model the behaviour that a variable, e.g., live-in variable, has to be assigned to this register file. Each input or output port has corresponding node in MRRG as well. Notably, since the register has storage functionality, any data goes into one input port can be delayed by several cycles before it is read from one output port. Therefore, the edge set between input node I and output node O can be defined as $E = \{(I_t, O_t), (I_t, O_{t+1}), (I_t, O_{t+2}), \ldots\}$. In other words, a register file has abundant routing resources to route signal along the time axis. Fig. 6c and 6d show the conversions of multiplexer and bus respectively, which are rather straightforward. By this abstraction, all the routing resources, whether they are physical or virtual ones, are modeled in same way using nodes and edges. This abstraction greatly reduces the complexity of the scheduling algorithm by exposing it an unified view of the architecture.

6 Modulo Scheduling Algorithm

Modulo scheduling is one of many software pipelining techniques [15]. Its objective 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- or 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 attack this problem for both unified and clustered VLIW [15, 16, 19, 20]. However, they can't be applied to the case of a coarse-grained reconfigurable matrix because the nature of the problem becomes more difficult.

6.1 Definition of Problem

For unified VLIW, scheduling means to decide *when* to place operation. For clustered VLIW, we also have to decide *where* to assign the operation, this is a placement problem. For coarse-grained reconfigurable architecture, there is one additional task: determining *how* to connect placed operations. This is essentially a routing problem. If we view time as one dimension of P&R space, the scheduling can be defined as a P&R problem

in 3D space, where routing resources are asymmetric and modulo constraints are applied. Table 6.1 shows a simple comparison among these modulo scheduling algorithms and the FPGA P&R problem.

	Sched.	Place	Route	Modulo
Unified VLIW	Yes	No	No	Yes
Clustered VLIW	Yes	Yes	No	Yes
Reconf. Arch.	Yes	Yes	Yes	Yes
FPGA's P&R	No	Yes	Yes	No

Table 1: Comparison between modulo scheduling for coarse-grained reconfigurable architecture and other related problems

Formally, the scheduling problem can be defined as mapping dataflow graph $G1 = \{V, E\}$ obtained from analysis and transformation on the MRRG graph $G2 = \{V; E, II\}$ defined at section 5. All MRRG nodes have to be used no more than once except artificially created ones. All the edges E of $G1$ have to be legally routed on $G2$. No resource constraints can be violated.

6.2 Algorithm Description

This scheduling problem is more complex, especially if the nature of P&R space and scarce routing resources are considered. In FPGA's 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 scheduling 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 which can forsee the routability during the routing phase. Our solution is to solve these two sub-problems in one framework. The algorithm is described in fig. 7.

Like other modulo scheduling algorithms, the outermost loop tries successively larger II, starting with an initial value equal to MII, until the loop has been scheduled. For each II, it first generates an initial schedule which respects dependency constraints, but may overuse resources. For example, more than one operation may be scheduled on one FU at the same cycle. In the inner loop, the algorithm iteratively reduces resource overuse and tries to come up with a legal schedule. At every iteration, an operation is riped up from the existing schedule, and is placed randomly. The connected nets are rerouted accordingly. Then a cost function is computed to evaluate the new placement and routing. A *simulated annealing* strategy is used to decide whether we accept the new placement or not. If the new cost is smaller than the old one, the new P&R of this operation will be accepted. Even if the new cost is bigger, there is still a chance to accept the move, depending on *"temprature"*. This method helps to escape from local minimum. The temperature is gradually decreased from a high value. So the operation is increasingly difficult to move. The cost function is constructed by taking account into overused resources. The penalty associated with them is increased every iteration. In this way, placer and router would try to find alternatives to avoid congestion. This idea is borrowed from the *Pathfinder* algorithm [18]. In the end, if the algorithm runs out of time budget without finding a valid schedule, it starts with the next II. This algorithm is time-consuming. It takes minutes to schedule a loop of medium size.

7 Experimental Results

7.1 Experiment Setup

To test the DRESC compiler, we employ a built-in architecture that resembles the organization of Morphosys [1]. In this configuration, a total of 64 FUs are divided into four tiles, each of which consists of 4x4 FUs. Each FU is not only connected to the 4 nearest neighbor FUs, but also to the FUs within 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 difference with Morphosys. In Morphosys, the architecture consists of a general-purpose processor and a reconfigurable matrix. Our test architecture is a convergence of a VLIW procesor and reconfigurable matrix. The first row of FUs can work as a VLIW processor with support of a multiport register file. Whenever the pipelined code is executed, the first row work cooperatively 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 two-fold. First, since the FUs in a VLIW processor and reconfigurable matrix are similar, we can reuse a lot of resources such as FUs and memory ports. Second, 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 4 programs, which are

```
II   := MII;

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

  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);

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

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

      if get a valid schedule then
        return scheduled;
    endfor

    if run out of time budget then
      break;

    UpdateOverusePenalty();
    UpdateTemperature();

  endwhile
  II++;
endwhile
```

Figure 7: Modulo scheduling algorithm for coarse-grained reconfigurable architecture

all derived from C reference code of TI's DSP benchmarks [21]. The *idct* is a 8x8 inverse discrete cosine transformation. The *fft* refers to a radix-4 fast Fourier transformation. The *corr* computes 3x3 correlation. The *latanal* is a lattice analysis function. They are typical multimedia and digital signal processing applications with abundant inherent parallelism.

7.2 Schedule Results

The schedule results are shown in table 2. The second column refers to the total number of operations within pipelined loops. The MII is the computed minimal initiation interval, whereas II 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 $IPC/No.of FUs$. It reflects the actual utilization of all FUs, excluding those used for routing. The last column is the CPU time to compute the schedule on a Pentium 4 1.6GHz PC.

kernel	no. of ops	MII	II	IPC	sched. density	time (sec.)
idct	86	2	3	28.7	44.8%	162
fft	70	3	3	23.3	36.5%	891
corr	56	1	2	28	43.8%	100
latanal	12	1	1	12	18.8%	5.2

Table 2: Schedule results

The IPC is impressively high, ranging from 12 to 28.7. It is well above any typical VLIW processor. The FU utilization are around 40% except for those kernels constrained by MII, e.g., *latanal*. The CPU time to calculate the schedule is relatively long because it is essentially a P&R algorithm. The II can't reach the ideal MII in some cases. One observation is that there are a large number of live-in and live-out variables around those pipelinable loops, which imposes pressure on the communication bandwidth between VLIW's register file and the rest of the matrix. IMPACT tends to generate hyperblocks with more live variables but less operations since it targets VLIW architectures, which have less FUs but a big unified register. To address this problem, the frontend algorithm has to be adapted to coarse-grained architectures.

8 Conclusion 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 really exploit the potential of coarse-grained reconfigurable architectures, big problems to solve are what kind of parallelism to exploit and how to extract it. We view loop-level parallelism as a key to outperform VLIW and superscalar processors. An automatic design or compiling tool is therefore needed.

We developed a retargetable compiler, DRESC, to attack this problem. A family of compiler-friendly coarse-grained reconfigurable architectures are proposed to enlarge the scope of pipelinable loops and eliminate the bottleneck of pipelining. An abstract machine representation, MRRG, is designed to transform a concrete architecture description to a unified abstract graph used by the modulo scheduling algorithm. To transform a program to a scheduling-ready dataflow

graph in an efficient way, various analysis and transformation techniques are implemented. A new modulo scheduling algorithm resembling placement and routing algorithms for FPGAs is developed. The results show up to 28.7 IPC and 44.79% FU density for tested kernels, proving the great potential for both coarse-grained reconfigurable architecture and the DRESC compiler.

However, the DRESC is still in its early phase of development. We are working to complete the unfinished parts, i.e., the simulator and the architecture parser, of entire flow. By then, we will be able to collect extensive data and do architecture exploration. We are also going to enhance the analysis and transformation techniques to further enlarge the scope of pipelinable loops, and improve the quality of our scheduling algorithm.

References

[1] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. on Computers*, vol. 49, pp. 465–481, May 2000.

[2] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor (abstract)," in *FPGA*, p. 261, 1998.

[3] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157–166, 1996.

[4] *Chameleon Systems Inc.* http://www.chameleonsystems.com.

[5] I. PACT XPP Technologies. www.pactcorp.com.

[6] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," in *Proc. International Workshop on Field Programmable Logic*, 1998.

[7] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a RAW machine," in *Architectural Support for Programming Languages and Operating Systems(ASPLOS-VIII)*, pp. 46–57, 1998.

[8] T. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, (San Jose, CA), ACM, 2000.

[9] P. F. C. Ebeling, D. Cronquist, "Rapid - reconfigurable pipelined datapath," in *Proc. of International Workshop on Field Programmable Logic and Applications*, 1996.

[10] W. Hwu, R. Hank, D. Gallagher, S. Mahlke, D. Lavery, G. Haab, J. Gyllenhaal, and D. August, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1640, Dec. 1995.

[11] R. Hartenstein, M. Hertz, Th.Hoffmann, and U. Nageldinger, "KressArray Xplorer: A new CAD environment to optimize reconfigurable datapath array architectures," in *ASP-DAC*, pp. 163–168, 2000.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *25th Annual Intl. Symp. on Microarchitecture*, pp. 45–54, 1992.

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pp. 266–275, 1991.

[14] J. W. Sias, W. mei W. Hwu, and D. I. August, "Accurate and efficient predicate analysis with binary decision diagrams," in *Proceedings of the 33rd annual IEEE/ACM international symposium on Microarchitecture*, pp. 112–123, 2000.

[15] B. R. Rau, "Iterative modulo scheduling," tech. rep., Hewlett-Packard Lab: HPL-94-115, 1995.

[16] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, and J. Eckhardt, "Lifetime-sensitive modulo scheduling in a production environment," *IEEE Transactions on Computers*, vol. 50, no. 3, pp. 234–249, 2001.

[17] M. S. Lam, "Software pipelining: an effecive scheduling technique for VLIW machines," in *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 318–327, 1988.

[18] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns, "Placement and routing tools for the Triptych FPGA," *IEEE Trans. on VLSI*, vol. 3, pp. 473–482, Dec. 1995.

[19] M. M. Fernandes, J. Llosa, and N. P. Topham, "Distributed modulo scheduling," in *HPCA*, pp. 130–134, 1999.

[20] C. Akturan and M. F. Jacome, "CALiBeR: A software pipelining algorithm for clustered embedded VLIW processors," in *ICCAD*, pp. 112–118, 2001.

[21] *TI Inc.* http://dspvillage.ti.com/.