Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling

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 compilation tools are essential to their success. In this paper, we present a modulo scheduling algorithm to exploit loop-level parallelism for coarse-grained reconfigurable architectures. This algorithm is a key part of our 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 utilization on tested kernels.

1 Introduction

Coarse-grained reconfigurable architectures have become increasingly important in recent years. Various architectures are proposed [19, 16, 2, 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 configuration storage space, flexible topology, combination with a general-purpose processor, etc. On the other hand, compared with traditional "coarse-grained" 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 electronics, 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 [19, 5] 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 [4, 12] focus on instruction-level parallelism (ILP) in limited scope, fail to make use of the coarse-grained architecture efficiently and in principle can not reach higher parallelism than a VLIW. Some recent research starts to exploit loop-level parallelism (LLP) by applying pipelining techniques [3, 9, 18], but still suffers from severe limitations in terms of architecture or applicability (see section 5).

To address the above problem, this paper presents a modulo scheduling algorithm, which is a key part of our DRESC framework[15], 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. 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. We test the algorithm on a set of benchmarks, and results show high performance and efficient resource utilization on an 8x8 coarse-grained architecture.

1530-1591/03 $17.00

The paper is organized as follow. Section 2 introduces target architectures. Section 3 covers various aspects of the modulo scheduling, illustrates the problem we face, presents our abstract representation of architectures and describes the algorithm in details. Section 4 reports the results and discusses current limitations. Section 5 discusses related work. Section 6 concludes the paper.

2 The Target Architecture

Our target platforms are 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 approach is similar to the work of KressArray [8]. 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, 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 [14], 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 provide small local storage space. The configuration RAM controls how the FU and multiplexors are configured, pretty much like instructions for processors. A few configurations are stored locally to allow rapid reconfiguration. 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 multiplexors. 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 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.

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 buses are used to connect tiles within the same row and column. Using this template we can mimic many coarse-grained architectures found in literature and also perform architecture exploration within the DRESC design space.

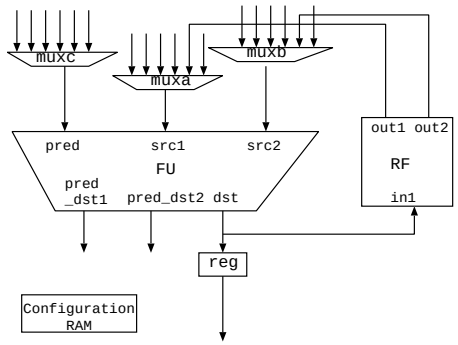

Figure 1. Example of FU and register file

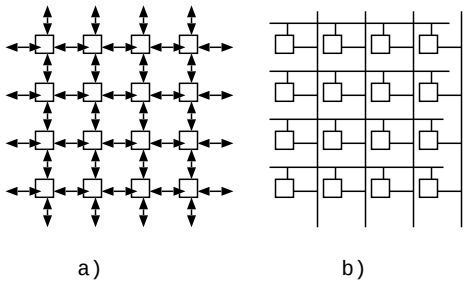

Figure 2. Examples of interconnection

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 *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 [17, 13, 7, 1]. However, they can not 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, let's consider a simple dependency graph, representing a loop body, in fig. 3a and a 2x2 matrix in fig. 3b. The scheduled loop is depicted in fig. 4a, where the 2x2 matrix is flattened to 1x4 for convenience of drawing, nevertheless, the topology remains the same.

Fig 4a is a space-time representation of the scheduling space. From fig. 4a, we see that modulo scheduling on coarse-grained architectures is a combination of 3 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

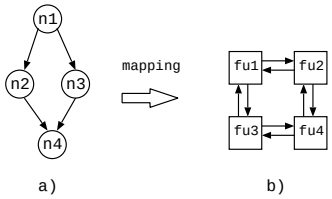

Figure 3. a) A simple dataflow graph; b) A 2x2 reconfigurable matrix

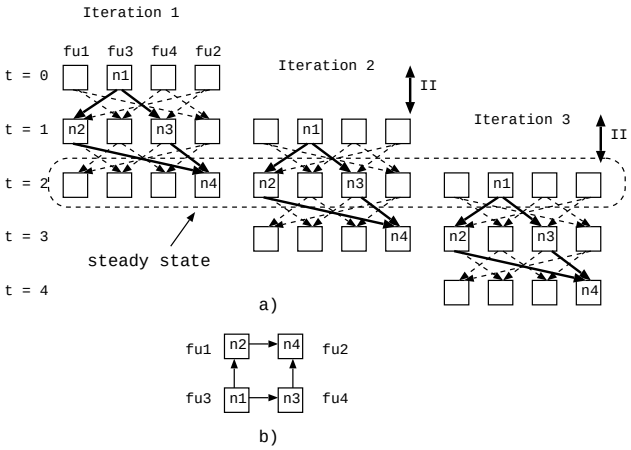

Figure 4. a) Modulo scheduling example; b) Configuration for 2x2 matrix

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. 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 2x2 matrix is shown in fig. 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.

3.2 Modulo Routing Resource Graph

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) [11] for software pipelining and the *routing resource graph* [6] 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 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 $E = \{(v_m, v_n) | t(v_m) <= t(v_n)\}$ corresponds to switches that connect these nodes – the restriction $t(v_m) <= t(v_n)$ modeling 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 3.3, this asymmetric nature imposes big constraints on the scheduling algorithm.

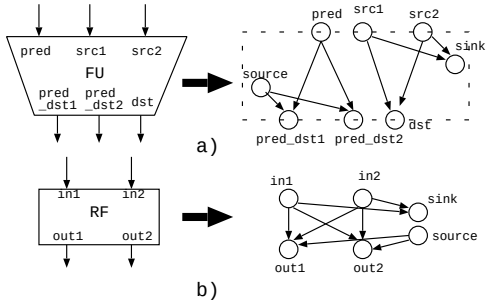

Figure 5. MRRG representation of DRESC architecture parts

During scheduling we start with a minimal II and iteratively increase the II until we find a valid schedule (see section 3.3). 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. Fig. 5 shows some examples. Fig. 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 axis. For FU, all the input and output ports have corresponding nodes in the MRRG graph. Virtual edges are created be-

tween 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. 5b shows a MRRG subgraph for a register file. Similar to a 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 a corresponding node in the MRRG as well. Notably, since the register has storage functionality, any data that 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 $E = \{(I_t, O_{t+1}), (I_t, O_{t+2}), \ldots\}$. In other words, a register file has abundant routing resources to route signals along the time axis. Other types of components can be modeled in a similar way. By this abstraction, all routing resources, whether physical or virtual, are modeled in the same universal way using nodes and edges. This unified abstract view of the architecture greatly reduces the complexity of the scheduling algorithm.

3.3 Modulo Scheduling Algorithm

The modulo scheduling algorithm takes a data dependency graph (DDG) representing the loop body and a MRRG representing the architecture as inputs. We use the IMPACT compiler framework [10] as a frontend to parse C source code, do some optimization and analysis, construct required hyperblock, 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 [15], e.g., a new method of removing prologue and epilogue code. Other transformations are borrowed from the VLIW compilation domain.

Although it is pointed out that our scheduling problem is indeed a P&R problem, it is more complex than traditional FPGA P&R, especially when the modulo and asymmetric nature of the P&R space and scarce routing resources are considered. 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 almost impossible to find a placement algorithm and 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. 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);
          accepted := EvaluateNewPos();  (4)
          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 run out of time budget then
      break;

    UpdateOverusePenalty();
    UpdateTemperature();

  endwhile
  II++;
endwhile
```

Figure 6. Modulo scheduling algorithm for coarse-grained reconfigurable architecture

First all operations are ordered by the technique in [13]. 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 [17].

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 rerouted accordingly. Then a cost function is computed to evaluate the new placement and routing (4). 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 *"temperature"*. This method helps to escape from local minima. The temperature is gradually decreased from a high value. So an operation becomes increasingly difficult to move. The cost function is constructed by taking into account overused resources. The penalty associated with them is increased every iteration. In this way, placer and router will try to find alternatives to avoid congestion. This idea is borrowed from the *Pathfinder* algorithm [6]. In the end, if the algorithm runs out of time budget without finding a valid schedule, it starts with the next II.

4 Experimental Results

4.1 Experiment Setup

We have tested our algorithm on an architecture that resembles the organization of Morphosys [19]. 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 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 support of a multi-ported register file. Whenever the pipelined code is executed, the first row works 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 all derived from C reference code of TI's DSP benchmarks [20]. 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.

4.2 Scheduling Results

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 1. Schedule results

The schedule results are shown in table 1. 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.

The IPC is 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 of its SA-based search strategy and computational cost of each iteration.

4.3 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. Second, at present it can not handle some architecture constraints, e.g., pipelined FUs and limited register files. Additionally, due to the way that the IMPACT frontend constructs the hyperblock for loop body, 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.

5 Related Work

Several research projects try to apply pipelining techniques to reconfigurable architectures in order to obtain high performance. RaPiD [2] has a linear datapath that is a different approach compared to 2-dimensional meshes of processing elements. This restriction simplifies application mapping but restricts the design space dramatically. Similarly, Garp [3] also features a row-based architecture allowing straight implementation of a pipeline. It doesn't support multiplexing, so the implementation is inefficient in case the II is bigger than 1. Recent work [9] tried to directly map loops to datapaths in a pipelined way. Lacking advanced scheduling techniques, it either uses a full-connected crossbar, or generates a dedicated datapath for several dataflow graphs, none of which is a good solution. PipeRench [18] 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, thus can not be applied to other coarse-grained reconfigurable architectures. Modulo scheduling algorithms on clustered VLIW architecture [1, 7] normally target a specific class of architectures and can not handle arbitrarily connected architectures. In addition, the routing problem is virtually unpresent or rather easy to solve in clustered VLIW architectures.

6 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 automatically.

We 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 28.7 IPC and 44.8% FU utilization for tested kernels, proving the potential for both coarse-grained reconfigurable architecture and our algorithm.

In the future, overcoming the limitations of the modulo scheduling algorithm and better integration into the DRESC design flow are our main focus.

References

[1] C. Akturan and M. F. Jacome. CALiBeR: A software pipelining algorithm for clustered embedded VLIW processors. In *ICCAD*, pages 112–118, 2001.

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

[3] 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, 2000. ACM.

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

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

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

[7] M. M. Fernandes, J. Llosa, and N. P. Topham. Distributed modulo scheduling. In *HPCA*, pages 130–134, 1999.

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

[9] Z. Huang and S. Malik. Exploiting operation level parallelism through dynamically reconfigurable datapath. In *DAC*, pages 337–342, 2002.

[10] The IMPACT group. http://www.crhc.uiuc.edu/impact.

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

[12] 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)*, pages 46–57, 1998.

[13] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Transactions on Computers*, 50(3):234–249, 2001.

[14] 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*, pages 45–54, 1992.

[15] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *International Conference on Field Programmable Technology*, 2002.

[16] 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*, pages 157–166, 1996.

[17] B. R. Rau. Iterative modulo scheduling. Technical report, Hewlett-Packard Lab: HPL-94-115, 1995.

[18] H. Schmit, D. Whelihan, A. T. an Matthew Moe, B. Levine, and R. R. Taylor. Piperench: A virtualized programmable datapath in 0.18 micron technology. In *Proc. of IEEE Custom Integrated Circuits Conference*, 2002.

[19] 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*, 49(5):465–481, May 2000.

[20] TI inc. http://dspvillage.ti.com/.