A Modulo Scheduling Algorithm for a Coarse-Grain Reconfigurable Array Template

Akira Hatanaka[1] and Nader Bagherzadeh[2]

[1]University of California, Irvine
Department of Electrical Engineering
and Computer Science
Irvine, CA 92697 USA
ahatanak@uci.edu

[2] University of California, Irvine
Department of Electrical Engineering
and Computer Science
Irvine, CA 92697 USA
nader@uci.edu

Abstract

Coarse Grain Reconfigurable Arrays (CGRAs) have been drawing attention due to its programmability and performance. Compilation onto CGRAs is still an open problem. Several groups have proposed algorithms that software pipeline loops onto CGRAs. In this paper, we present an efficient modulo scheduling algorithm for a CGRA template. The novelties of the approach are the separation of resource reservation and scheduling, use of a compact three-dimensional architecture graph and a resource usage aware relocation algorithm. Preliminary experiments indicate that the proposed algorithm can find schedules with small initiation intervals within a reasonable amount of time.

1. Introduction

In the past several years, sales of electronic gadgets, such as MP3 or DVD players and PDA communication devices, have achieved significant growth. Traditionally, these devices have been controlled by embedded microprocessors combined with a few hardwired accelerators to meet the stringent and conflicting requirements, such as low power consumption and high performance. However, as these embedded devices become more and more sophisticated and powerful, it is getting difficult to meet the demand for functionalities and computational power with the combination of microprocessors and hardwired accelerators.

CGRA [3] is a new category of architectures that has been drawing attention recently due to its programmability and computational power.

1-4244-0910-1/07/$20.00

Although many institutions, both from academia and industry, have proposed and developed various CGRA architectures, automatic compilation of a program written in a high level language is still a problem that remains unsolved. The difficulty of automatic compilation onto CGRAs has its roots in the micro-architectural features of CGRAs that are different from conventional general purpose microprocessors, i.e. distributed register files and shared interconnect architectures that are sparse and irregular. These features make it difficult to apply conventional compiler algorithms to CGRAs.

Several groups have proposed algorithms that automatically compile a program onto CGRAs. However, most of them either assume the underlying micro-architectures, including both the computational units and interconnect architectures, are homogeneous or of regular structures, to achieve short compilation time with a light heuristic,or try to compile onto an irregular architecture by a random algorithm, such as simulated annealing, at the expense of a very long compilation time.

In this paper, we propose a modulo scheduling algorithm that is capable of compiling loops in a program onto a family of heterogeneous CGRAs, which consists of processor elements and possibly irregular interconnect architectures. We propose a two phase algorithm consisting of a resource reservation phase and a scheduling phase. A compact graph representation of the target architecture is used to map the application. Also, a resource usage aware placement algorithm is used to shorten the time it to find a legal solution.

The organization of the paper is as follows. Section 2 presents related work. Section 3 explains the background of this work, including a brief discussion of software pipelining algorithm on CGRAs, followed by an explanation of the work in [8]. Section 4 explains the software tools used for this work. Section 5 discusses the proposed modulo

scheduling algorithm in this work.

Section 6 shows the results of compiling kernels of benchmark applications onto different CGRA architectures. Finally, section 7 concludes this paper.

2. Related Work

2.1 Reconfigurable Architecture

Many groups have developed coarse grain reconfigurable arrays over the last decade. [12] is a SIMD-like 8×8 mesh architecture. It broadcasts the same instruction word to all the processing elements in a column or a row, and can efficiently exploit regular data parallelism in a program. The architectures of [11] and [13] share the traits of dataflow machines. Operations are fired as soon as all the input data packets are available. [7] is a reconfigurable accelerator with distributed processor elements and register files controlled by very long instruction words.

2.2 Compilers for CGRAs

Compilation onto CGRAs has been a topic of active research during the last few years. Since compilation for CGRAs is much more complex than that for conventional general purpose processors, many works make certain assumptions about the target architecture. The works in [9], [14] and [6] all propose constructive scheduling heuristics for CGRAs, assuming some regularities in the interconnect architectures. The compilation problem becomes much simpler when some of the idiosyncratic micro-architectural features are removed.

The compiler in [14] uses a space-time scheduler that decouples the partitioning phase, or the placement phase, and the scheduling phase.

[6] adopts a one-pass algorithm that divides the application graph into clusters and schedules the operations in sequence.

However, both [14] and [6] are not capable of producing code for an architecture with irregular interconnect architectures. [14] is a homogeneous mesh architecture that uses a MIPS processor as its processing elements. Although the target architecture of [6] is parameterizable, the base architecture in the paper is a regular mesh.

Also they do not exploit instruction level parallelism beyond a basic block. Since the amount of parallelism available within a basic block is usually small, they cannot achieve a large speedup in many cases.

The work in [8] is closest in its goal to our work. The compilation algorithm modulo-schedules loops onto a CGRA architecture template. The graph representation of the architecture allows a wide range of architectures to be targeted.

To apply modulo scheduling to CGRAs, [8] uses a three-dimensional architecture graph representation that is generated by replicating the two-dimensional spatial architecture along the time axis. The problem becomes very similar to placement and routing for FPGAs[1].

The approach iteratively relocates operation vertices and reroutes communication edges until no resource overuses exist. The problem of this approach is that depending on the target architecture and application program to be scheduled, it may take a very long time to come up with a feasible schedule. The time axis of the three-dimensional architecture graph has to be at least as long as the number of cycles of the final schedule, which makes the size of the architecture graph large. Restricted relocations of operations can also contribute to longer running time as explained later in this paper.

3. Background

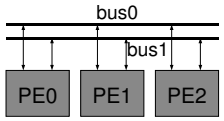
3.1 Software Pipelining

Software pipelining [5] improves performance by overlapping the execution of different iterations of a loop. An iteration can start executing before completions of previous iterations. The interval at which iterations are started is called the *initiation interval*, abbreviated II. The goal of the software pipelining optimization is to find a schedule that overlaps iterations and uses the shortest possible initiation interval.

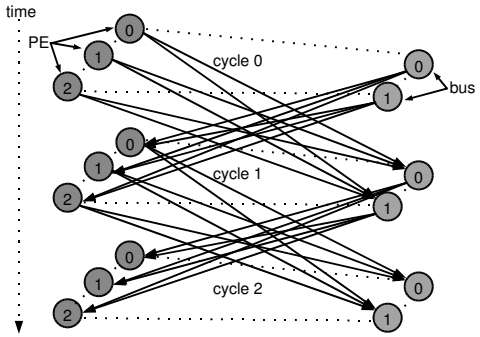
Many algorithms have been developed that effectively software pipeline loops. One of the well-known algorithm is Modulo Scheduling, first proposed in [10]. The approach adopts list scheduling with height-based operations priority. This approach is effective when the architecture has a centralized register file through which all functional units communicate, and any functional unit can freely read and write operands any time without interference from other functional units. For this type of architecture, the scheduler can assume the communication resources that are used to send data from one functional unit to another can be used freely.

For architectures with partitioned register files [2], the scheduler needs to pay attention to additional issues. The partitioning of operations determines the amount of communication placed on the inter-cluster communication bus. The scheduler needs to schedule operations so that there are no conflicts on the communication bus. Still, for a simple VLIW architecture with two register files, the scheduler can produce high quality codes using simple heuristics.

In addition to these issues, mapping onto CGRAs needs to address routing. Because there is a high degree of sharing of interconnect architecture resources between PEs, and it takes variable number of cycles to communicate data from

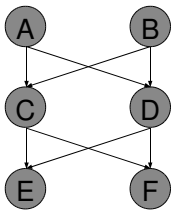


(a) Architecture



(b) Three-dimensional graph

Figure 1. Three-dimensional graph representation of an architecture



(a) Application graph

cycle	PE0	PE1	PE2	Bus0	Bus1
0	A	B			
1				A -> D	B -> C
2	C	D			
3				C -> F	D -> E
4	E	F			
5					

(b) Initial schedule

cycle	PE0	PE1	PE2	Bus0	Bus1
0	A				
1		B		A -> D	
2					B ->C,D
3	C				
4			D	C ->E,F	
5					D -> E
6		E			
7			F		

(c) Final schedule

Figure 2. Motivational example

one PE to another, the task of determining a route becomes much more complicated.

3.2 Motivational Example

Algorithms to modulo schedule loops onto CGRAs have been proposed recently by several groups. As mentioned in section 2, our work is close in its goal to [8], which places few restrictions on the interconnect architecture.

The approach in [8] borrows ideas from placement and routing algorithms developed for FPGAs. Similar to placement and routing algorithms for FPGAs, the approach represents the architecture and the application program as directed graphs. The vertices and the edges of the architecture graph together represent relevant features and interconnection of different hardware resources of the architecture, such as input/output ports of functional units, register files and multiplexers.

The uniqueness of this approach is that the architecture graph is a three dimensional representation, which has a time dimension in addition to the space dimensions. The three dimensional graph is built by replicating the vertices of the two dimensional spatial representation of the architecture along the time dimension, and adding edges accordingly. For example, the architecture in Fig.1(a) can be represented as the graph in Fig.1(b).

Representing architectures as three-dimensional graphs allows simultaneous scheduling, placement and routing of the application graph. The operation vertices are placed onto vertices in the architecture graph. Data dependency edges between these operations are routed using shortest path algorithms, with weights of architecture edges representing resource overuses. The whole problem becomes almost equivalent to finding a feasible placement and routing solution for a FPGA, except that there are additional constraints on resource usages due to the nature of modulo scheduling,

The problem of this approach is that adding the time dimension may not be effective in finding a feasible solution in a reasonably short amount of time.

First the sheer size of the graph may lengthen the time to find a legal solution. The graph has to be replicated along the time axis, at least as many times as the length of the final schedule. For example, if the final schedule takes twenty cycles, the replicated graph will have at least twenty times as many nodes as the spatial graph representation.

Second, the routability of dependency edges of an operation vertex are restricted by the vertex's predecessor and successor vertices' location and cycle. To clarify our point, we will use Fig.2 as an example.

Suppose the application graph in Fig.2(a) is to be mapped onto an architecture in Fig.1(a), and $II = 2$. An initial schedule may look like the one in Fig.2(b), which is clearly not legal and needs to be fixed. Since operation vertices must always adhere to their precedence constraints,

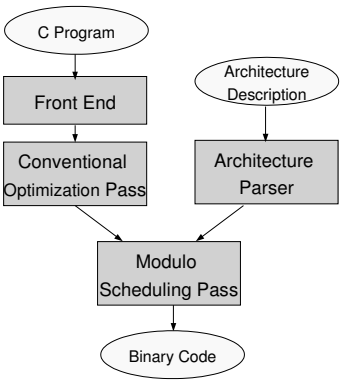


Figure 3. Framework of the modulo scheduler

vertices B,C and D cannot be relocated to their final locations unless their successors are moved to later cycles. For example, relocation of operation B to PE1 at cycle 1 will not be accepted, since vertex (PE0, cycle = 2), which is the location of vertex C, is not reachable from (PE1, cycle = 1). In other words, there are no routes in Fig.1(b) that starts from vertex (PE1, cycle1) and terminates at vertex (PE2, cycle2). Vertex B cannot be relocated because its successors C and D are scheduled too early, and C and D cannot be pushed down either, because of E and F. Of course, it is possible to include slacks in the schedule by extending the architecture graph along the time axis to give more freedom to the vertices to move around. However, this will result in even larger sizes of architecture graphs. Even if the graph was extended, the same situation could show up locally where movements of vertices are restricted by their predecessors and successors. A lot of attempts to relocate vertices will end up being rejected, and it will take a long time to find a legal solution.

As explained in later sections, we propose an approach to schedule loops without inflating the size of the architecture graph.

4. Tool Framework

Fig.3 shows the whole compilation framework. The framework takes in two files created by the user: the application program and the architecture description.

4.1 Application Program

The application program is written in C, with annotation pragmas specified by the programmer to indicate which portion of the code is going to be scheduled.

The current implementation of the mapping framework can only software pipeline the most inner loops of a program. In order to schedule multiply-nested loops, the programmer has to manually rewrite them into singly-nested loops. In the future, we plan to incorporate a pass into our framework that automatically or semi-automatically software pipelines multiply-nested loops using source level transformation techniques [5].

The application program first goes through the frontend pass. Then conventional compiler optimizations, such as constant propagation, copy propagation and common subexpression elimination, are applied to remove as many redundancies as possible. After that, the application program is transformed into a graph representation, and passed to the modulo scheduling pass. Information on loop invariants and immediate constants is conveyed as well. This information is necessary to generate initialization code that loads loop invariants before the initiation of the first loop iteration.

4.2 Architecture Description

Similar to the work done in [8] and [4], our work aims to target a wide range of CGRAs.

The target architecture is defined in an architecture description file. The description includes functional unit specifications, register file specifications, local memory specifications and the interconnect architecture specification.

Although we do not discuss it in this paper, functional units can be heterogeneous: different functional units can execute different sets of operations of various latencies. Functional units are connected to register files to read their operands. For simplify scheduling, we assume that when a functional unit reads its operands from a register file, there are no conflicts of accesses to resources. That is, each functional unit has exclusive access to one or more register file read ports.

Register files can have different number of registers and read/write ports. All the operands used by functional units are read from register files: no functional unit can read directly from an output of another functional unit or interconnect network.

The interconnect architecture consists of registers and multiplexer. A variety of interconnect topologies, such as meshes, rings, buses and trees, can be constructed by a combination of multiplexers and registers. Multiple-hop communications are also allowed.

The connections between components explained above are specified as a netlist. Information on the individual components and the connections between them is sufficient to generate graphs used for modulo scheduling and designs in the form of HDL modules.

An example of a target architecture model is shown in Fig.4. The architecture consists of Processor Elements (PEs) and interconnect networks through which PEs send

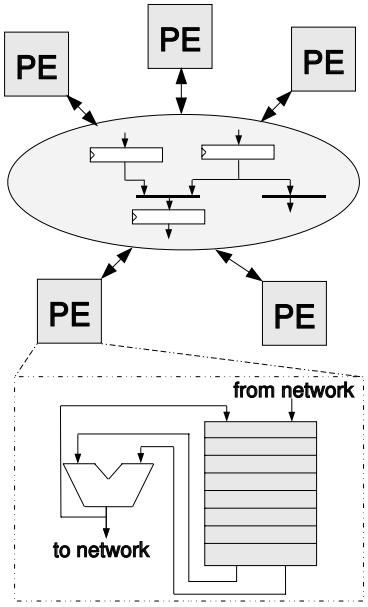


Figure 4. Example target architecture

and receive data to and from each other.

5. Modulo Scheduling Algorithm

In the following sections, the compilation framework and the algorithms to modulo-schedule loops onto CGRAs are explained.

5.1 Internal Data Structures

5.1.1 Application Graph

The vertices of the application graph represent operations. The edges of the graph are various types of dependencies including data dependencies and precedence dependencies.

5.1.2 Architecture Graph

As briefly mentioned in section 3, our modulo scheduling algorithm consists of two phases.

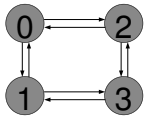
- **Resource reservation phase**
  The first phase assures the resources necessary at the steady state, i.e. functional units consumed by operations and routing resources consumed by communications, are available.

- **Scheduling phase**
  Based on the result of the first phase, the second phase assigns cycles to each operations.

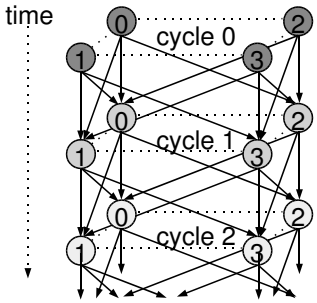
Because we do not try to do everything from placement & routing to scheduling in one pass, we do not have to use a three-dimensional architecture graph that is replicated as many times as the length of the schedule, as explained in [8].

Instead, we use an architecture graph that is only II times as large as the original two-dimensional graph. In this way, we can keep our approach scalable.

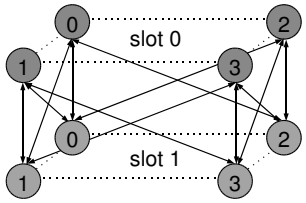
Fig.5 compares the difference between the architecture graphs adopted in this work and in [8].



(a) 2D graph of the example architecture



(b) Work in [8]



(c) This work

Figure 5. Comparison of architecture graphs

The two-dimensional representation of an example architecture is shown in Fig.5(a). The vertices represent processing elements, which include inside computational and storage units, while the edges represent communication resources. Each edge has a latency of one clock cycle, e.g. it takes one clock cycle to send data from vertex 0 to vertex 1.

Fig.5(b) and Fig.5(c) show respectively the three-dimensional representation proposed in [8] and this paper. The algorithm to generate the architecture graph in Fig.5(c) is shown in Fig.6.

First, all the nodes in the original two-dimensional representation are replicated II times ($II = 2$, in the example). The edges are added between vertices that belong to differ-

```
procedure GenArchGraph
  for each u_i ∈ vertices of orgG
    for j = 0 to II − 1
      add vertex v_{i·II+j} to newG
  for each e ∈ edges of orgG
    v_s ← source(e)
    v_t ← target(e)
    for j = 0 to II − 1
      if (e consumes a cycle) then
        add edge (v_{s·II+j}, v_{t·II+(j+1)%II}) to newG
      else
        add edge (v_{s·II+j}, v_{t·II+j}) to newG
  for each u_i ∈ vertices of orgG
    if (u_i is an input port of an RF) then
      for j = 0 to II − 1
        add edge (v_{i·II+j}, v_{i·II+(j+1)%II}) to newG
end procedure
```

Figure 6. Pseudo code of the algorithm to generate architecture graphs

ent slots, if the edge in the original graph consumes a cycle. Otherwise, edges are added between vertices that belong to the same slot. Finally, edges that connect input ports of register files are added.

5.2 Algorithm Overview

The algorithm borrows ideas from research on FPGA placement & routing algorithms. Vertices of the application graph are relocated and edges are rerouted to iteratively decrease the overuses of resources. The algorithm is based on simulated annealing, and the process is repeated until a feasible solution is found. For this work, we assume the architectures have sufficient number of registers to schedule the application without having to spill some operands to the local memory. Also, we assume the primary goal of modulo scheduling is to find schedules with small IIs and smaller IIs result in shorter schedules.

The outermost loop of the algorithm slowly decreases the temperature. The pseudo code of the inner loop of the algorithm is shown in Fig.7.

The operation vertices are popped from queue $appVQ$ one at a time, and relocated to another functional unit. The new location is chosen based on how much resources will be used as a result of relocation. compCost() is obtained by computing the number of overused resources. Details are explained in later sections. A vertex that is already placed on the new location will be evicted, and relocated to the original location of the vertex that caused the eviction.

Outgoing edges of the relocated vertices and direct predecessors of the relocated vertices are rerouted using Dijkstra's shortest path algorithm. The algorithm is run once per vertex. The formula to compute edge weights is given below:

```
procedure InnerLoop
  appVQ ← all vertices of application graph
  while appVQ is not empty do
    v ← pop appVQ
    oldCost ← compCost()
    if (SetNewLocation(v) = true) then
      routeEdges()
    newCost ← compCost()
    if (accept(newCost − oldCost) = false) then
      restore relocated vertices
  end while
end procedure
```

Figure 7. Pseudo code of the inner loop of the mapping algorithm

$$weight(e) = cycleWeight(e) + ruWeight(e) \quad (1)$$

where, $cycleWeight(e)$ and $ruWeight(e)$ are given as follows:

$$ruWeight(e) = \begin{cases} 0, \ if\ e\ unoccupied \\ W1 \times use(e), \ otherwise \end{cases} \quad (2)$$

$$cycleWeight(e) = \begin{cases} W2, \ if\ e\ consumes\ cycle \\ 0, \ otherwise \end{cases} \quad (3)$$

The edge weight is the sum of two parts: the cycle weight, $cycleWeight$, and the resource usage weight, $ruWeight$.

The cycle weight, W2, is given to resource edges that consumes a cycle, such as registers, and biases the algorithm to choose paths with smaller total number of cycles.

The resource weight is proportional to the number of application edges that are assigned to the resource edge, which is denoted as $use(e)$. The weight factor, W1, is increased after every annealing iteration, if the resource overuse on the resource edge doesn't get resolved.

After relocating vertices and rerouting edges, the overall cost, which is the sum of the number of overuses of resources is computed. Then the algorithm decides whether to accept the perturbation, based on how much the overall cost has changed. If the solution is not accepted, the vertices and the edges are restored to their original state.

5.3 Resource Usage Aware Placement

From our experiments, we found that a large portion of the running time is spent on running Dijkstra's shortest path algorithm when vertices are relocated. We also found that a lot of times, relocating operation vertices to randomly chosen new locations doesn't decrease the overuse of resources. Therefore, instead of randomly choosing a new location for each vertex, we choose new locations that are likely to lead to decrease in resource usages. The pseudo code of the function that chooses a new location is given in Fig.8.

```
procedure SetNewLocation(v)
  found ← false
  initialize n
  while (found = false || n > 0) do
    oldLoc ← save old location of v
    newLoc ← select new location of v
    if (a vertex already placed on newLoc) then
      ev ← vertex placed on newLoc
    else
      ev ← NULL
    newCost = GetPlacementCost(newLoc, v, oldLoc, ev)
    oldCost = GetPlacementCost(oldLoc, v, newLoc, ev)
    if (acceptPlacement(newCost - oldCost)) then
      found ← true
      break
    n ← n − 1
  end while
  return found
end procedure

procedure GetPlacementCost(locV, v, locEV, ev)
  vSet ← {v, ev} ∪ {direct predecessors of v and ev}
  sum ← 0
  for each v ∈ vSet
    patialSum ← 0
    for each e ∈ outedges of v
      c ← numberOfCycles(v, target(e))
      partialSum ← partialSum + c
    sum ← sum + partialSum/√outDegree(v)
  return sum
end procedure
```

Figure 8. Pseudo code of the algorithm to find a new location

The procedure $SetNewLocation$ repeatedly searches for a new location for a vertex v that is likely to decrease the resource usages. It compares the resource usage estimates obtained by the procedure $GetPlacementCost$ to determine whether or not to accept the new location. If it cannot find a new location after a predetermined number of attempts, it gives up and returns a null location.

The procedure $GetPlacementCost$ uses the sum of the number of cycles the rerouted edges consume. For source vertices with multiple out edges, the sum is divided by the square root of the out degree, to take into account the possible resource sharings among the out edges.

6. Experimental Results

All the experiments were run under linux 2.4, on a 1.6GHz Pentium 4 processor with 512MB of memory.

The benchmark applications used in this work are shown in Table.1. The second column shows the number of operation vertices in the application graphs.

	size
DCT	87
FFT	118

Table 1. Benchmark applications

For the experiments, we targeted three architectures. The first one is a 4×4 mesh-like architecture with additional interconnect resources that connect functional units in the same row or column, as seen in Morphosys[12]. The second one is a 8×8 mesh-like architecture that is constructed by replicating the 4×4 architecture. The third one is a tree-like architecture that has four clusters, each of which consisting of four PEs. The PEs in a cluster communicate with each other through three local buses. The number of communications between PEs belonging to different clusters is limited to four, two for incoming communications and two for outgoing communications.

The experimental results are shown in the table below.

	8×8, II=3			8×8, II=2		
	w	wo	util	w	wo	util
DCT	29	110	45	55	323	68
FFT	69	98	61	143	na	92

(a) 8x8 architecture, with different IIs

	4×4 mesh			8×8 mesh			tree		
	w	wo	util	w	wo	util	w	wo	util
DCT	3.3	8.4	90	55	323	68	46	na	90
FFT	4.0	16.1	92	143	na	92	56	69	92

(b) Running time when the minimum IIs were targeted

Table 2. Experimental results

The columns "w" and "wo" are the program running time in seconds, with and without resource aware placement

respectively. The columns "util" shows the utilization of the functional units in percentage.

Table.2(b) shows that, when resource aware placement was turned on, the proposed algorithm was able to find legal solutions with the minimum IIs for all combinations of target architecture and benchmark application in a short amount of time. Although it is difficult to make a fair comparison, since our target architectures differ from those in [8], the shorter running time of the program and the tighter schedule achieved underscore the effectiveness in our approach.

In some cases, when resource aware placement was turned off, the program wasn't able to come up with a legal solution within the given amount of time. The results clearly show that resource aware placement played a large role in shortening the program running time. On average, resource usage aware placement reduces the running time by 65%.

Table.2(a) shows how much the running time varies when the target II was changed. As expected, running times increase when the target II is smaller.

7. Conclusion

This work proposes a modulo scheduling algorithm that can target CGRAs with irregular interconnect architectures. The algorithm uses a compact three-dimensional architecture graph onto which application graphs are mapped. A placement algorithm that is aware of possible resource usages is proposed to reduce the program running time.

References

[1] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns. Placement and routing tools for the triptych fpga. *IEEE Trans. VLSI Syst.*, 3:473–482, 1995.

[2] E. Granston, E. Stotzer, and J. Zbiciak. Software pipelining irregular loops on the tms320c6000 vliw dsp architecture. *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 138 – 144, 2001.

[3] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 642 – 649, March 2001.

[4] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array architectures. *Proceedings of the ASP-DAC 2000. Asia and South Pacific Design Automation Conference, 2000.*, pages 163–168, January 2000.

[5] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. *In Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[6] J. Lee, K. Choi, and N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Des. Test. Comput.*, 20(1):26–33, January 2003.

[7] J. Leijten, G. Burns, J. Huisken, E. Waterlander, and A. van Wel. Avispa: a massively parallel reconfigurable accelerator. *International Symposium on System-on-Chip, 2003. Proceedings*, pages 165 – 168, November 2003.

[8] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Design, Automation and Test in Europe Conference and Exhibition, 2003, Proceedings*, 2003.

[9] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2006*, October 2006.

[10] B. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, 1994.

[11] T. Sato, H. Watanabe, and K. Shiba. Implementation of dynamically reconfigurable processor dapdna-2. *VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT)*, pages 323 – 324, April 2005.

[12] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers*, 49(5):465–481, 2000.

[13] M. Vorbach and R. Becker. Reconfigurable processor architectures for mobile phones. *Proceedings of Parallel and Distributed Processing Symposium, 2003.*, April 2003.

[14] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86 – 93, September 1997.