Pipeline Synthesis and Optimization for Reconfigurable Custom Computing Machines[*]

Markus Weinhardt
Universität Karlsruhe, Fakultät für Informatik
D-76128 Karlsruhe, Germany
(weinhard@ipd.info.uni-karlsruhe.de)

January 3, 1997

Abstract

This paper presents a pipeline synthesis and optimization technique for high-level language programming of reconfigurable Custom Computing Machines. The circuit synthesis generates hardware accelerators from a sequential program which exploit the reconfigurable hardware's parallelism. Program loops are transformed to structural hardware specifications. The optimization algorithm uses integer linear programming to balance and pipeline the circuit's registers. This global optimization determines the minimal amount of flip-flops necessary for an optimal pipeline throughput. It also considers the irregular flip-flop distribution on FPGAs. Standard interface circuitry and a runtime system provide the connection between the accelerator unit and its host computer. An integrated compiler invokes the synthesis and produces a program which downloads, calls and controls its hardware accelerators automatically.

1 Introduction

Reconfigurable Custom Computing Machines (CCMs) have proven useful for many applications. They combine the flexibility of software with the speed of application-specific hardware. The program part executed in software takes advantage of the universality of a general-purpose processor. Yet portions executed in hardware can be accelerated enormously. However, programming CCMs remains a difficult task since tradeoffs between software and hardware must be considered, and the circuits accelerating the application must be designed manually. This work aims at *automatically* extracting and generating accelerators from a sequential (software) program.

The following section presents the components of our high-level CCM compiler. Next, section 3 details the flowgraph synthesis which generates circuits from the program's FOR-loops; and section 4 introduces a new optimization algorithm for pipelining. Finally, we discuss related work, report results, and draw some conclusions.

2 High-level language compilation for CCMs

This section describes the main aspects of our high-level programming approach for CCMs. We analyze a program written in a sequential programming language (as C or MODULA-2) and extract hardware accelerators for it. The target architecture is a standard host computer with a field-programmable accelerator unit comprising FPGAs and local memory.

[*]This work has been supported by the Deutsche Forschungsgemeinschaft, Graduiertenkolleg "Beherrschbarkeit komplexer Systeme" (GRK 209/2-96).

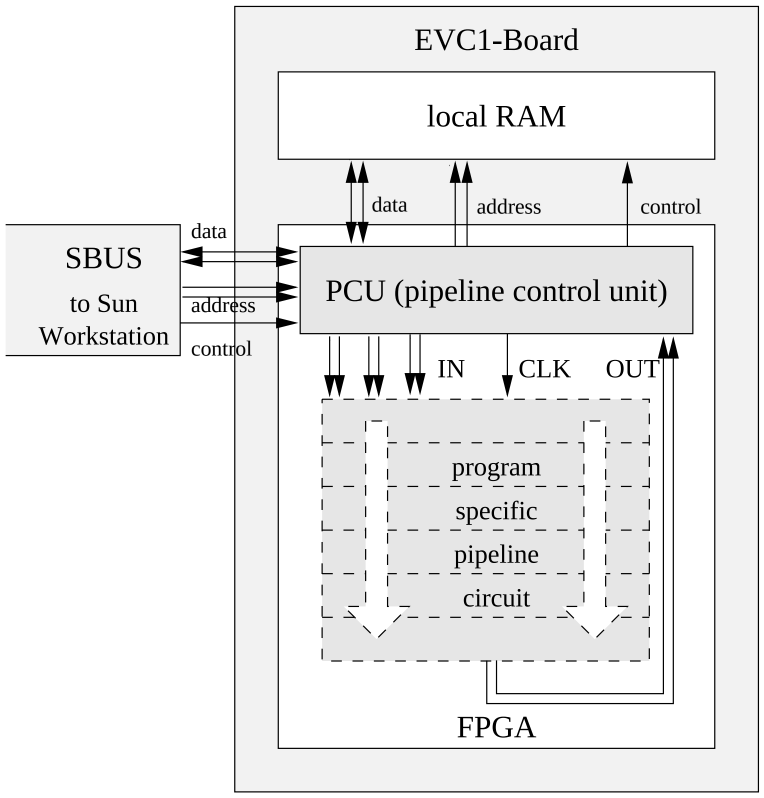

Figure 1: EVC1 with Pipeline Control Unit

We consider FOR-loops as hardware candidates since they express iterative computations. And computations which perform the same operations on a large set of data are most likely to benefit from hardware acceleration. The loops used for hardware synthesis must be in the following normal form: The loop bodies may not contain function calls or inner loops, and the FOR-statement must have the structure `FOR I:=0 TO N DO ...`, i. e. the loop counter starts with `0` and is always incremented by one. Finally, index expressions in the loop body may not depend on variables defined in the loop. If not given, this normal form can often be achieved by source language transformations (function inlining, loop unrolling, induction variable substitution). These transformations are state of the art [1, 2] and therefore not repeated here.[1]

The pipeline synthesis and optimization algorithm (detailed in sections 3 and 4) generates flowgraphs for FOR-loops in normal form. These flowgraphs are then instantiated with hardware components from an operator library. Using this library, we can also give estimates on the resources required by a pipeline and on the expected speedup for implementing the loop in hardware. This information is used by a subsequent *partitioning* algorithm which automatically determines if a loop should be executed in software or in hardware. This partitioning can also be performed dynamically at run time. Then the actual loop length and the configuration state of the CCM are considered, too.

To allow automatic operation, the program specific pipelines are controlled by a *pipeline control unit (PCU)*. Figure 1 shows the PCU in the experimental environment we use, a Sun SPARCstation with Virtual Computer Corp.'s EVC1 board. The PCU is used for every program and controls the pipeline operation. I. e. it accesses and stores data, fills and flushes the pipeline and controls the

[1]Additionally, external operating system or library calls, pointer operations and — due to the limitations of current FPGA technology — floating-point operations cannot be synthesized to hardware.

communication of the accelerator unit with the host computer. The PCU buffers input and output values internally. So it provides more input and output ports to the pipeline than there are local memory banks. However, due to the sequential memory access, more ports decrease the pipeline throughput.

Another advantage of the PCU is the fact that it makes the pipeline independent of the actual hardware and thus facilitates porting the compiler. It also allows the software to call standard functions for sending and receiving data to and from the local memory and for operating the pipeline. Together with runtime-system functions which configure and reset the FPGAs, these functions allow to automatically download and call the accelerators. [3] presents more details on the PCU and the hardware/software interface.

3 Flowgraph synthesis

This section describes how dataflow graphs (or flowgraphs for short) are synthesized from a FOR-loop. The flowgraph will be used as a pipeline, i. e. execute the loop's iterations in an overlapped fashion. Therefore loop-carried dependencies (which restrict the loop's parallelism) have to be realized as feedback cycles in the flowgraph.

3.1 Acyclic flowgraph generation

First, we apply compiler optimizations as constant propagation and common subexpression elimination [1] to the loop body. This reduces the pipeline size. Then we analyze the loop body as if it was executed only once. A method similar to the transmogrifier C compiler `tmcc` [4] is used. It analyzes the dependencies of the statements and creates a purely combinational, acyclic flowgraph.[2] Conditional statements (the only control construct allowed in the normal form) are implemented by multiplexers, and array accesses are treated in the same way as scalar variables.

The flowgraph's input nodes have to be initialized with the values valid at the loop entrance, and the values of output nodes must be read after loop execution. To enable pipeline processing, array input and output nodes are realized as ports which provide (or process, respectively) the arrays as sequential data streams.[3] So we can treat an index shifted version of a one-dimensional input array (e. g. `X[I-2]` with loop variable `I`) as a *delayed* version of another access of the same array (e. g. `X[I]`). A shift-register (including the intermediate values, here `X[I-1]`) saves the old input values and therefore reduces the I/O bandwidth requirements of the flowgraph. Additionally, the direct flow of intermediate values to the next operator saves cycles for storing and loading these values. Figure 2 shows a loop (which we will use as a running example) and its flowgraph. The shaded nodes represent the input shift-register.

```
TMP := 0;
FOR I := 0 TO N DO
   X[I]   := TMP + Y[I+1];
   Y[I+1] := X[I] + Y[I] - Y[I] / 8;
   TMP    := X[I];
END
```

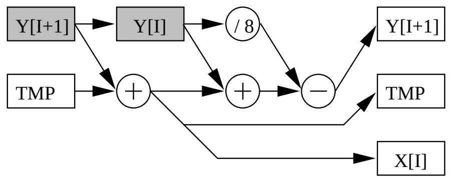

Figure 2: FOR-loop and its flowgraph

[2]In contrast to `tmcc`, we create a flowgraph on the word level rather than the bit level.

[3]Therefore, we currently only allow array accesses of the form `X[I+C]` and `X[-I+C]` (`C` constant). Arrays must be read and written in the same direction to allow overlapping of the read, process and write phases. However, we can treat multi-dimensional arrays if their higher-dimension indices do not depend on the loop variable and have *always* different values at runtime. Then, their one-dimensional subarrays are treated as different one-dimensional arrays.

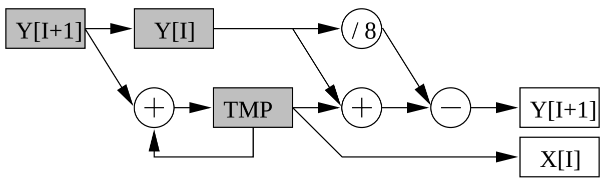

Figure 3: Flowgraph with feedback cycle

```
X[-1] := 0;
FOR I := 0 TO N DO
   X[I]   := X[I-1] + Y[I+1];
   Y[I+1] := X[I] + Y[I] - Y[I] / 8;
END
```

$\Rightarrow$

```
X[-1] := 0;
TMP := X[-1];
FOR I := 0 TO N DO
   X[I]   := TMP + Y[I+1];
   Y[I+1] := X[I] + Y[I] - Y[I] / 8;
   TMP    := X[I];
END
```

Figure 4: Loop with array dependency and its transformation

3.2 Feedback cycles

In order to guarantee correct execution of the pipeline, we have to analyze loop-carried dependencies and accordingly introduce *feedback cycles*. These dependencies exist if values defined in a loop iteration are used in a subsequent iteration. This is the case if a scalar variable is an input and an output in the flowgraph. Then, we add a register to the output node and feed its value back to the input node. The register holds its initial value only for the first loop iteration and stores the feedback value on each successive iteration. Figure 3 shows the flowgraph with a feedback cycle for register TMP (shaded). A multiplexer, along with select and clock enable logic, for choosing and storing the correct value in the register is necessary, too, but not shown in the figure. Variables with mutual dependencies may also lead to feedback cycles with more than one register.

A dependency for an array exists if the output node's index is the input node's index incremented by one (if the array is traversed in increasing order) or decremented by one (if the array is traversed in decreasing order).[4] This is not the case in our example loop. Though array Y would prevent parallelization, the access order of its elements is correct in a pipeline. So the flowgraph of figure 3 is correct. A flowgraph with an array feedback cycle is very inefficient because it uses only a few values from the input port. Most of the time the values fed back from the output node are used. This results in a large waste of I/O bandwidth. Therefore, we perform a high-level loop transformation which introduces new scalar variables for the few initial values. Then, we can use the method for scalar feedbacks and save input ports. Figure 4 shows a loop with an array dependency and its transformation — our example loop. The transformation substitutes the new scalar variable TMP for X[I-1].

Our detailed analysis decides for all loops if they can be executed by a pipeline or not. However, as we will see in section 4.1, large feedback cycles will result in poor pipeline throughput. This can reduce the attainable speedup significantly.

4 Optimal pipelining

The flowgraphs generated in section 3 are not always correct. The registers inserted in the feedback cycles delay the values on some paths from the input to the output nodes. For correct execution the delays on all paths must be equalized by *register balancing*. On the other hand, paths without feedback cycles have a very long combinational delay. They should be *pipelined* to increase

[4]Here we consider all the registers of an input shift register as input nodes.

throughput. Therefore, we must perform register balancing and pipelining together:

> **Register insertion problem**
> Find a correct circuit which allows maximal pipeline throughput using the minimal number of FPGA flip-flops.

We do not optimize the pipeline's latency because the time for filling and flushing the pipeline hardly affects the overall performance. And the PCU (cf. section 2) is able to handle varying latencies automatically.

4.1 Clock period computation

The attainable clock period T_C can be determined before performing the pipelining itself. Therefore, it is used as a fixed parameter of the pipelining method.

The clock period T_C is determined by the number of input and output ports in most cases. On a CCM with one memory bank it is computed by

$$T_C \quad := \quad N_I \times T_I + N_O \times T_O$$

where N_I and N_O are the numbers of input and output ports, respectively, and T_I and T_O are the times for reading and writing a local memory word, respectively. T_I and T_O depend on the speed of the used RAM and the available clock frequencies. $T_I = 50\ ns$ and $T_O = 100\ ns$ on the EVC1 board we use.

In some cases, very large feedback cycles will increase the required clock period. But it is not allowed to insert a register in a cycle because it would change the circuit's functionality. However, we can reduce the clock period by optimally distributing the registers in cycles with two or more registers.[5] We do not consider further high-level optimizatons as those proposed in [5].

Other reasons for an increased clock period are the operator delays. But unless the operators are part of a feedback cycle, we can always choose pipelined implementations for the operators themselves. So they will not increase the clock period and thus reduce the throughput.

4.2 ILP for optimal register insertion

Solving an *integer linear program (ILP)* determines integer variable values which minimize a linear *cost function* according to a set of linear *constraints* (inequalities). The next sections give a new formalization of the register insertion problem as an ILP. Then, we can use the simplex and branch-and-bound algorithms to solve this global combinatorial optimization problem efficiently.

We formally consider the flowgraph $G = (N, E)$ as a set of nodes N and a set of edges $E \subset N \times N$. Furthermore, $I \subset N$ is the set of input nodes, $O \subset N$ the set of output nodes, $AI \subset I$ the set of array input nodes, and $P \subset N$ the set of pseudo operators which contain no logic (e. g. constant shifts).

4.2.1 Preprocessing

We have to preprocess the flowgraph before we can extract constraints for the ILP from it. First, the feedback cycles are replaced by single *supernodes* because their registers must remain fixed. This yields a directed acyclic graph (DAG). The *node latency* NL_i of the supernodes is set to the number of registers in the cycle. We define $NL_i = 0$ for purely combinational operaters, and NL_i equals the number of internal registers for pipelined operators.

[5]In this case one register has to remain at the cycle's exit, and we have to adjust the initialization of the moved registers.

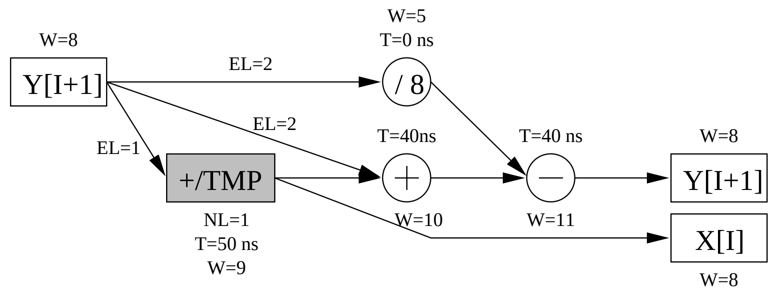

Figure 5: Preprocessed flowgraph

Input values	
T_C	clock period (in ns)
NL_i	node latency of node i
$EL_{i,j}$	edge latency of edge (i,j)
W_i	output width of node i
$T_{i,j}$	signal propagation time (in ns) from output of node i to output of node j
Computed values	
d_i	maximal delay from register to output of node i (in ns)
r_i	number of registers inserted at output of node i
s_i	number of registers saved by merging with operator i
l_i	latency with respect to array input nodes (in clock cycles)

Table 1: Notation

Next, the shift registers which realize the delayed array inputs have to be removed because they are subject to optimization. Instead, an edge from the array's input node to the node where the delayed input was used is added. Its *edge latency* $EL_{i,j}$ is set to the required number of delays. $EL_{i,j} = 0$ for all other edges. Figure 5 shows G for our example. (Assume $NL = 0$ and $EL = 0$ unless otherwise stated.)

4.2.2 Notation

Table 1 defines the required notation. T_C, NL_i and $EL_{i,j}$ have been explained in the previous sections. The operator's output width W_i is used to determine the precise number of flip-flops needed, and $T_{i,j}$ represents the combinational delay of operator j with respect to its input from node i. For nodes j representing feedback cycles or pipelined operators, $T_{i,j}$ is the delay from the input on edge (i,j) to the first internal register. Since we cannot accurately estimate routing delays, we add a constant average routing delay to all values $T_{i,j}$. To guarantee a working circuit, $T_{i,j} \leq T_C$ must hold for all edges (i,j). Figure 5 shows these input values, too.[6]

The following computed values are all non-negative integers. d_i is used as intermediate value to keep track of the accumulated delay of an operator chain. r_i and s_i count the required registers and are used in the cost function. Finally, l_i is the number of registers on a path from an array input port to node i, i. e. its latency. Because l_i determines the number of registers inserted on all paths to node i, it guarantees a balanced pipeline.

[6]There is only one T value given for every node since the values are the same for all incoming edges.

4.2.3 Cost function

The solution of the ILP minimizes the number of inserted flip-flops. Thereby we automatically unite register chains at different outgoing edges of a node. The flip-flop count is represented by the following cost function:

$$C = \sum_{i \in N} r_i \cdot W_i - \sum_{i \in N \setminus I \setminus O \setminus P, NL_i = 0} s_i \cdot W_i$$

The first term computes the flip-flop count of all nodes by summing the products of r_i (number of registers needed at a node i) with the register's width W_i. The second sum computes the flip-flops which can be saved by merging them with the operator's combinational logic ($0 \leq s_i \leq 1$). It only applies to operators which contain logic but no internal registers. This merging is possible in FPGA families which combine combinational logic and flip-flops in a logic block. For other families, we simply omit the second sum.

4.2.4 Constraints

The following contraints define admissible solutions:

For all input nodes i, the delay from a register is 0:

$$\forall i \in I : d_i = 0 \tag{1}$$

For all array input nodes i, the latency from array inputs is 0:

$$\forall i \in AI : l_i = 0 \tag{2}$$

All input and output nodes i must be registered:

$$\forall i \in I \cup O : r_i > 0 \tag{3}$$

The number of registers saved by merging is limited by 1 and the number of registers instantiated at all:

$$\forall i \in N : s_i \leq 1, s_i \leq r_i \tag{4}$$

The accumulated delay of any node must not exceed the clock period:

$$\forall i \in N : d_i \leq T_C \tag{5}$$

The DAG edges order the operator execution: Thus, for all edges (i, j), the latency of node j must be at least as large as that of node i plus the internal latency of node i:

$$\forall (i, j) \in E : l_j \geq l_i + NL_i \tag{6}$$

The number of registers at a node's output is determined by its own and its successors' latencies. Thus, for all edges (i, j), r_i is larger than or equal to the difference of node latencies plus the edge latency minus the internal latency of node i:

$$\forall (i, j) \in E : r_i \geq l_j - l_i + EL_i - NL_i \tag{7}$$

For edges (i, j) with no register inserted ($l_i = l_j$), the accumulated delay of node j is at least the sum of that of node i and the propagation time $T_{i,j}$. For registered edges, no constraint for d_j applies:[7]

$$\forall (i, j) \in E : d_j \geq T_{i,j} + d_i + T_C \cdot (l_i - l_j) \tag{8}$$

The delay of node j is at least the maximum of its incoming edges' propagation times:

$$\forall (i, j) \in E : d_j \geq T_{i,j} \tag{9}$$

[7]For registered edges, $d_i = 0$ and $l_i - l_j < 0$. Since $T_{i,j} \leq T_C$ always holds, $T_{i,j} + d_i + T_C \cdot (l_i - l_j) \leq 0$.

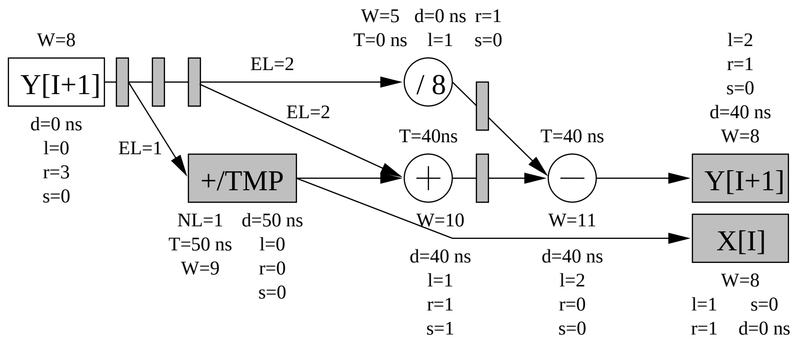

Figure 6: Flowgraph with computed values and pipeline registers

4.2.5 Register insertion

After solving the ILP, we insert registers in the flowgraph in the following way: First, add r_i registers to the output of every node i; and next, replace all edges (i, j) by edges from the n-th register to node j, where $n = l_j - l_i + EL_{i,j} - NL_i$. This automatically combines the registers in all outgoing edges of a node. Figure 6 shows the resulting flowgraph. The values were computed using the mixed IP–solver [6]. In this example, the algorithm actually has to consider the bitwidths of the operators to decide between inserting a register at the output or at the input of the `/8`-node. Otherwise, it could not determine the minimal number of flip-flops.

5 Related work

Some previous work on generating hardware accelerators from a software program has been performed. For example, the PRISM system [7] extracts coprocessors from C functions. However, it does not exploit hardware parallelism on a higher level as our pipeline synthesis does. On the other hand, Guccione et al. [8, 9] use vector code to synthesize operator pipelines similar to ours. This enables more hardware parallelism but requires the programmer to write programs in less general vector code.

Our loop analysis is similar to methods used in vectorizing and parallelizing compilers [2]. They allow nested loops with arbitrary strides for the array accesses, but cannot detect all loops which could be parallelized.

Several methods have been proposed for optimal register insertion and balancing. For example, [10] discusses pipelining algorithms for vector computers, and [11] proposes an optimal balancing technique using ILP for data flow computers. But both methods assume a machine architecture with fixed register width and standard delay for all operators. None of these assumptions are true for pipelines implemented on FPGAs. Therefore we have extended [11] to integrate balancing and optimal pipelining for FPGAs.

6 Results and Conclusions

We presented new pipeline synthesis and optimization techniques for high-level CCM programming. Experiments with a prototype MODULA-2 compiler have shown the general feasibility of our compilation approach. Earlier measurements achieved speedups up to 14.1 for a FIR-filter application, and up to 21.1 for a greyscale-image smoothing application [3]. The experiments compared software runtime on a Sun SPARCstation 10 to hardware configuration and execution time on an EVC1 board.

The example loop of section 3 shows that our dependence analysis handles a considerably larger class of loops than parallel loops. It automatically synthesizes scan (or prefix) operators which otherwise have to be specified explicitly, e. g. in parallel SIMD languages. Hence, we can synthesize more efficient hardware accelerators for standard programs.

The ILP formalization introduced here computes the exact number and placement of flip-flops necessary for optimal throughput of pipelines implemented in FPGAs. Heuristic approaches cannot do this exactly. This optimization on the flip-flop level, rather than on the register level, is especially necessary for coarse-grained FPGAs which have relatively few flip-flops per combinational gate. It enables higher resource utilization in high-level CCM programming.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.

[3] M. Weinhardt. Portable pipeline synthesis for FCCMs. In *Field-Programmable Logic and Applications; 6th International Workshop*, pages 1–13. Springer-Verlag, September 1996.

[4] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, Napa, CA, April 1995.

[5] P. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, 1981.

[6] M. Berkelaar. Unix manual page of `lp_solve`. Eindhoven University of Technology, Design Automation Section, 1992.

[7] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, March 1993.

[8] S. Guccione. *Programming Fine-Grained Reconfigurable Architectures.* PhD thesis, University of Texas at Austin, May 1995.

[9] S. A. Guccione and M. J. Gonzalez. A data-parallel programming model for reconfigurable architectures. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87, Napa, CA, April 1993.

[10] K. Hwang and Z. Xu. Multipipeline networking for compound vector processing. *IEEE Transactions on Computers*, 37:33–47, January 1988.

[11] G. R. Gao. Algorithmic aspects of balancing techniques for pipelined data flow code generation. *Journal of Parallel and Distributed Computing*, 6:39–61, 1989.